GPU acceleration of the frequency domain acceleration search for binary pulsars

Christopher Laidler

A Thesis Presented for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

UNIVERSITY OF CAPE TOWN

February 2020

Supervised by

Associate Professor M.M Kuttel

Abstract

Graphics processing units (GPUs) have been used to accelerate computation in a broad range of fields; this work presents a GPU-accelerated search for pulsars. Pulsars are highly magnetised neutron stars with extremely stable rotational periods. These periods can be accurately measured, which makes them exceptionally powerful reference tools in the field of astrophysics. Pulsars have very weak emissions, making them difficult to find. Most pulsars are found in large-scale surveys, which generate a large amount of data, and require extensive data processing. This work describes a GPU-based solution, with implications for real-time processing of pulsar search data.

Pulsar astronomy uses radio telescope observations with high spectral and temporal resolution, which produce very large data sets and require intensive Digital Signal Processing. Large-scale pulsar surveys using next-generation radio telescopes such as the Square Kilometre Array (SKA), will have to be performed in real time as the volumes of raw data produced will be too large to be stored for an extended period. These computational requirements are compounded when searching for binary pulsars as their orbital motion makes them difficult to detect using classic periodicity searches. However, these rare pulsars are of great interest to physicists, as they allow us to test general relativity.

Acceleration searches are the most common technique for detecting signals from binary pulsars that may be missed by standard search techniques. One of these, the frequency domain acceleration search (FDAS), mitigates the effect of orbital acceleration by correlating a matched template with the spectrum of a signal. This method has been shown to be more efficient than the alternative time domain acceleration search (TDAS)s. Even so, it is extremely computationally intensive to perform on a large scale. The existing implementation, `Accelsearch`, is run on a central processing unit (CPU), which limits its performance.

We address this problem by creating a GPU port of the FDAS. An analysis of the fundamental calculations on which the FDAS is based informs the design of a fully asynchronous pipeline that exploits multiple levels of parallelism. This entails developing a novel technique for calculating Fresnel integrals, which increases the speed and numerical accuracy of the calculations, in both single- and double-precision. Furthermore, we develop a new estimate which improves the numerical accuracy of filter coefficients for accelerations close to zero. The GPU-accelerated pipeline achieves speeds 30 to 70 times faster than the existing serial CPU implementation.

Our results clearly show that GPU acceleration is effective at reducing the cost of processing the FDAS component, to the point at which the SKA1-mid survey data could be searched in real time using 340 to 675 desktop GPUs from the Pascal generation.

Acknowledgements

I would like to thank my supervisor, Associate Professor Michelle Kuttel, for her guidance and assistance in structuring my thesis. For his insight into the world of pulsar astronomy that gave me the foundation to undertake this work, I am very appreciative to Scott Ransom. Thank you also to Scott for showing a fellow climber the ropes on distant shores. Not being associated with a pulsar research group, I am indebted to Scott and the NRAO for providing me with some invaluable data. The financial assistance of the SKA South Africa towards this research is hereby acknowledged, and greatly appreciated. For the patience and understanding of my employers and colleagues at HealthQ, I am very grateful.

I would not have been able to complete this journey without the support of my friends and family. For many years of feeding me, clothing me and putting up with my shenanigans, I am eternally grateful to my parents, Gigi and Dennis, and my brother Nicholas. I have relied on the friendship and support of too many people to name. Thank you for putting up with my eternal absenteeism. In particular, my troupe of greasy mechanically-minded friends, you bring the colour into my life. A further word of thanks to the other postgrads with whom I've lived and shared many a hard time: Meg, Steve, Margaux, Carlos, Marion, and Hannes. Thanks for all the shared meals and shoulders to cry on. Last, but most definitely not least, I am forever indebted to my partner, Grace. Without you, my work would be truly unreadable. You have brought the words to make my technical skills shine.

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.

2. I have used the IEEE transactions convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been attributed, and has been cited and referenced.

3. This thesis is my own work.

4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Signature Signed by candidate

Date 10/02/2020

Table of Contents

Chapter 1

Introduction

Graphics processing units (GPUs) have become prevalent in high-performance computing (HPC) and have been used to accelerate computationally intensive processes in a wide range of fields. This work describes the design, implementation, and benchmarking of a GPU-accelerated search used to find pulsars, specifically those in tight binary orbits with other stellar objects.

Pulsars are highly magnetised rotating neutron stars that emit beams of electromagnetic (EM) radiation from the regions above their magnetic poles. Since their discovery over fifty years ago, the study of pulsars has contributed a rich wealth of insight into the fields of physics and astronomy [59, 68, 69]. Pulsars were used both to confirm the existence of gravitational radiation [127, 128] and to place constraints on the equations of state of neutron matter [28]. Pulsars, especially those in binary orbits, have proven to be excellent laboratories for a variety of tests of gravitational theories [59, 62]. There are many theories that deviate from classic general relativity (GR), many of the limits by which these theories can deviate have been set by observing pulsars [3, 16, 40, 62]. Further, a double pulsar system is one of the only known ways to test strong-field GR [57, 70]. Indeed, many of the areas of fundamental physics can currently be best, or even only, studied by observing pulsars [117].

One of the current objectives of pulsar science is the study of gravitational waves. These waves have the potential to reveal valuable information about the structure and nature of the universe [21]. They were first predicted by Einstein's theory of GR and have only recently been directly measured using ground-based laser interferometers [1]. Low-frequency gravitational waves (such as those generated from pairs of merging supermassive black holes [48]) can be detected and studied with the use of a pulsar timing array (PTA) [39]. A PTA is a collection of 20 or more millisecond pulsars (MSPs), which are regularly and accurately timed over several years. For the reasons outlined above, there is great interest in finding and studying new pulsars – particularly those in binary orbits.

1.1 Observing Pulsars

Pulsars are generally discovered and observed with radio telescopes. Pulsar emissions consist of a stream of extremely regular pulses of EM radiation. These emissions occur across a wide range of the EM spectrum, and have been detected in the radio, optical, X-ray, and gamma-ray bands. The radiative flux density of most pulsars decreases with frequency: the spectra of most pulsars ($\sim$79% [49]) exhibit a simple power law with a mean spectral index of about -1.6 [49]. However, there are a number of factors that hinders the observation of pulsars at low frequencies ($\lesssim 300\,\mathrm{MHz}$). Thus, the majority of pulsars have been discovered and are studied with radio telescopes operating at frequencies in the range $300\,\mathrm{MHz}$ to $2\,\mathrm{GHz}$ [122].

The broadband nature of the emission introduces an additional complicating factor in the form of dispersion. As a pulse propagates through the interstellar medium (ISM), the EM waves interact with ionised particles and cause a frequency-dependent delay. This dispersion causes the pulse to be observed on Earth at higher frequencies before lower frequencies. When observing pulsars, this dispersion needs to be compensated for through a process known as dedispersion.

There are two main types of observation associated with pulsar astronomy: *search observations* and *timing observations*. The latter type comprises targeted observations of known pulsars, which are only possible with the most sensitive radio telescopes [39]. The primary objective of a timing observation is to measure the time of arrival (TOA) of an individual pulse [8, 9]. However, the emissions from pulsars are generally weak, with the signal from individual pulses usually falling below the level of background radio noise. Therefore, a pulse can only be accurately identified through the periodic nature of a pulsar signal. This requires long-duration observations that span tens of thousands to hundreds of thousands of individual pulses [115, 124]. Pulsar observations usually cover a broad bandwidth, and have very high spectral and temporal resolutions [69]. Thus, these observations generate a large volume of raw data, which has to be processed by a pulsar search backend[1]. These require a commensurate amount of digital signal processing (DSP) to distill out the important information such as pulse TOAs [31].

Search observations are intended to find new pulsars, and require similar or more sensitive radio telescopes than those used for timing observations. In fact, most new pulsars are discovered by

[1]The custom hardware and software used to process data from a telescope.

only the largest radio telescopes in the world [76]. Due to the higher sensitivity requirements of search observations, their durations are often longer than that of timing observations. They require similar, or higher, spectral and temporal resolutions, and thus produce larger volumes of data [124]. In addition, searches are often conducted as large-scale surveys that involve up to thousands of *pointings* [124]. These large data sets need to be searched for faint periodic signals with unknown properties such as spin period and dispersion. Pulsar searching requires DSP that has to explore a large parameter space and each unknown parameter covered by the search exponentially compounds the basic DSP requirements.

The past fifty years of research in this field have proven the value of pulsar astronomy [59, 68, 69]. However, the next decade will see an unprecedented expansion of the field, facilitated by the next generation of radio telescopes – the Five-hundred-meter Aperture Spherical Telescope (FAST) and the Square Kilometre Array (SKA). The SKA will be one of the largest telescopes ever built and will be a factor of 10 to 100 times more powerful than the current generation of radio telescopes [18, 121]. One of the primary science goals of the SKA is to find the majority of the observable pulsars in our galaxy [117]; this would represent an anticipated ten-fold increase in the number of known pulsars [54, 58]. The volume of data that will be produced by the SKA during pulsar searches is so large that it prohibits storage of the raw data for any extended period [66]. Thus, the entire pulsar search pipeline[2] will have to be processed in real-time[3] [66], presenting a tremendous computational challenge.

1.1.1 Acceleration searches

Since binary pulsars interact with other stellar objects, they allow a wider range of measurements and are therefore of greater scientific value. However, binary pulsars with short orbital periods are especially challenging to find. This is due to the change in relative velocity caused by the orbital motion, which Doppler shifts the EM pulses, causing a modulation of the observed spin period. This modulation adds the unknown orbital parameters to the search parameter space; the computational intensity of even the most basic pulsar search is so great that computation has been a limiting factor for many large-scale surveys [11, 56, 68, 87, 107, 115–118]. As computational

[3]A pipeline is a process model in which the components are connected in series, where the output of one component is the input of the next one.

[3]The duration of the observation.

power has increased over time, new techniques and hardware have allowed observations from several early surveys to be reprocessed [32, 33, 55, 81, 84]. This reprocessing has lead to the discovery of many new pulsars in old data. Thus, HPC has become an integral part of the search for new binary pulsars and is the focus of this work.

In the last two decades, a large proportion of HPC has moved away from the classic central processing unit (CPU) to a variety of alternative hardware architectures. One type of hardware in particular, the GPU, which is a discrete coprocessor based on a highly parallel architecture, has gone from fringe to mainstream [44]. The power, flexibility, and cost of Nvidia's GPUs– combined with their Compute Unified Device Architecture (CUDA) application programming interface (API) – has made them one of the most prominent players in the HPC arena [36]. This hardware comes with significant considerations, however. When porting codes to the GPU architecture there is often some consideration of the potential trade-off between the speed at which certain computations are performed, and their numerical accuracy [63]. These trade-offs usually entail the selection of which floating-point precision to use for performing specific key computations, and whether to use the special high-speed, reduced-accuracy intrinsic functions [91] available on GPU architectures.

By far the most widespread and successful means of searching for pulsars in tight binary orbits is acceleration searching [20, 51, 83, 115], which can detect periodic signals that have a constant change in frequency. In most circumstances, search observations cover only a short portion of the orbit of a binary pulsar. In these cases, the modulation of the observed spin frequency is approximately linear, meaning that an acceleration search is sufficient to correct for the Doppler shift caused by the orbital motion. Acceleration searches are divided into two main categories. The first operates in the time domain (resampling a time series prior to Fourier transforming it [20]) while the second category operates in the frequency domain (correlating an acceleration filter with the discrete Fourier transform (DFT) of the signal [115]). The latter is more computationally efficient, as each trial filter can be applied with several short independent fast Fourier transforms (FFTs), whereas the former requires an FFT of the full time series for each acceleration trial.

The frequency domain acceleration search (FDAS) operates by creating sections of two-dimensional (2D) power spectra and searching these for pulsar candidates. The 2D power spectra are created

by convolving a number of matched filters with the spectrum of the signal. The coefficients of these *acceleration* filters are described in terms of the Fresnel integrals. Performing the many convolutions of this method is very computationally intensive, due to the calculation of these integrals. There exists a highly successful and widely used CPU implementation of the FDAS, known as `Accelsearch`, which is part of the pulsar search package PRESTO [112]. The FDAS exhibits a large degree of parallelism, which is not exploited by the serial `Accelsearch` application.

The majority of the key DSP components of the pulsar timing and search pipelines have been ported to run on some form of parallel architecture. The noteworthy exception to this is the FDAS; at the time of writing, no massively parallel implementation of the FDAS was available. In the pulsar search pipeline, the FDAS is a clear next candidate for GPU acceleration. Reducing the run time of this computationally intensive component can drastically reduce the overall computation time and thus the cost of running a large-scale pulsar search. The focus of this work is exploring this computational challenge.

1.2 Statement of the problem

At the onset of this project, there was no massively parallel implementation of the FDAS – the most efficient of the acceleration searches. Further, there was no published analysis of the speed, precision, and accuracy of the computations upon which the FDAS method is based. A detailed analysis of these computations has the potential to reveal optimisations which can increase either the speed, the accuracy, or both. Integrating these optimisations in a GPU-accelerated FDAS would drastically reduce the time and cost of future surveys. This highlights a clear need for an efficient parallel implementation of the FDAS, as well as an analysis of the computation involved.

1.3 Aims

The primary aim of this work is to reduce the time and cost of performing a pulsar acceleration search, without compromising its sensitivity. We endeavour to accelerate the search to the extent that the region of the parameter space where most known pulsars are found, can be covered in real time. This real-time processing can be achieved through two methods: by parallelising the execution of the search, and by tailoring the numerical accuracy and precision of the calculations

to meet the desired sensitivity. We aim to analyse the FDAS pipeline in order to determine where parallelism can be exploited, and use this to design and implement a GPU-accelerated pipeline that exploits multiple levels of parallelism. We further aim to analyse the numerical accuracy of the FDAS calculations in order to increase the efficiency of our implementation, and to present this analysis.

A secondary aim is to make the new functionality compatible with existing software, to allow easy integration of this work into existing pulsar search backends. Furthermore, the implementation should be highly configurable to allow flexibility and close-to-optimal performance across a wide range of architectures and search parameters.

1.4 Approach

In setting out to achieve the aims outlined above, we took a top-down approach: identifying the components of the existing implementation, drafting a parallel pipeline, implementing and optimising individual components, and putting together a fully asynchronous pipeline.

We began by profiling the existing CPU implementation; from this, two stages and a number of components were identified. These components were then ranked by the influence they have on the overall speed of the search. The purpose of this was to identify which would benefit from being run on the GPU. This allowed us to identify the fundamental calculations that are used in the FDAS. We then assessed the numerical accuracy of computing these values in both single- and double-precision, identifying several alterations which would increase the numerical accuracy.

Once we had unpacked the components, we drafted a pipeline which runs memory transfers, CPU computation, and GPU computation in parallel. This new asynchronous pipeline includes a number of changes that increase the speed and accuracy of the search. Notably, we changed the way in which the second stage of the search is performed, as the existing implementation is not well suited to being run on a massively parallel architecture. We then proceeded to implement this pipeline, first individually porting the relevant components to run on a GPU. During this process, each component was profiled and optimised, with a focus on the trade-offs between speed and accuracy. These components were then merged into an asynchronous pipeline, which aimed to maximise the utilisation of the CPU, GPU, and memory buses. Finally, the complete pipeline was benchmarked for speed and accuracy.

1.5 Research contribution

The primary contribution of this work is a software package called `AccelGPU`, a highly efficient port of `Accelsearch`, created to run on a GPU. `AccelGPU` accelerates all stages of the FDAS and exploits multiple levels of parallelism, allowing the workload to be balanced between the CPU and the GPU. We introduce a new technique for performing the optimisations in the second stage of the search, which increases both the speed and the accuracy of this stage. This effectively reduces both the run time and the cost of performing this acceleration search.

This work presents a detailed analysis of the trade-offs between the numerical accuracy and computational speed of the fundamental calculations upon which the FDAS is based. The most significant of these is the evaluation of Fresnel integrals; here, we introduce a new technique that efficiently increases the numerical accuracy of the calculation of these integrals in both single- and double-precision. Furthermore, we find that the numerical accuracy of the acceleration filter coefficients deteriorates rapidly as acceleration approaches zero. To mitigate this, we introduce a new, more accurate means of calculating acceleration filter coefficients for accelerations close to zero. These are all combined into a library that exposes a number of highly optimised functions to perform these calculations in both single- and double-precision.

1.6 Thesis organisation

The remaining chapters of this thesis are organised as follows. Chapter 2 briefly introduces pulsars, and describes techniques used in searching for new pulsars. A summary of the technical details of the existing serial implementation of the FDAS is given in Chapter 3 A review of general-purpose graphics processing unit (GPGPU) computation, specifically related to Nvidia GPUs and the CUDA API, is given in Chapter 4.

The decomposition of the FDAS pipeline is explained in Chapter 5. The purpose of this is to ascertain how to structure the new GPU-accelerated pipeline to enable multiple levels of parallel computation. This chapter begins with the profiling of the existing FDAS implementation and concludes with a description of the high-level design of the new GPU-accelerated FDAS search. Chapter 6 outlines the technical details of the implementation of the new pipeline. It provides a detailed analysis of the numerical accuracy of the individual components, as well as the effects of

a number of the low-level configuration parameters on these components. Chapter 7 reports on the overall performance of the asynchronous pipeline, run with the chosen default configurations. It is then followed by an analysis of the impact of key configuration parameters on the performance of the asynchronous pipeline. Conclusions are presented in Chapter 8, together with a discussion of future work.

Chapter 2

Pulsars and how to discover them

This chapter gives a brief outline of pulsars. It includes a description of the life cycle of a pulsar, which serves to delineate a number of the key pulsar sub-populations and characteristics. The main body of the chapter is a discussion on how new pulsars are found. This includes a brief overview of the full pulsar search pipeline, with details of a number of alternative techniques for one of its main components: the periodicity search. The aim of this is to familiarise the reader with the pulsar search ecosystem into which this work fits.

Pulsars – rapidly rotating neutron stars – were first observed by Jocelyn Bell and Anthony Hewish in 1967 [47]. In the subsequent 50 years, pulsars have contributed much to the fields of physics and astronomy. Neutron stars are formed in type II supernova explosions of intermediate mass (8–20 $M_\odot$) stars. In such a supernova, if the mass of the core exceeds the Chandrasekhar limit ($\sim$1.4 $M_\odot$ [23]), it may collapse under its own gravity, forming a neutron star. Neutron stars are predicted to have masses that range from 1.4–3 $M_\odot$ (Tolman–Oppenheimer–Volkoff limit [103]) with radii in the order of 12 km. This makes them the densest known stars [61], surpassed only by black holes. The drastic reduction in size resulting from the collapse of the star's core significantly increases the rotational rate through the conservation of angular momentum. This results in rotational periods of between 14–140 ms. In addition, the core collapse increases the surface magnetic field strength to the order of 10^{11}–10^{13} G. When neutron stars were first theorised in the 1930s [7], it was thought that they would be too small and dim to be observed directly. However, the pulsating radio signal discovered in 1967 by Jocelyn Bell was correctly attributed to a rotating neutron star, ushering in the era of pulsar astronomy.

The pulsar emission mechanism is a complex electrodynamic process occurring in the pulsar magnetosphere. This mechanism is not yet fully understood, however the somewhat simplified *lighthouse model* [42, 104, 110] (Figure 2.1) is commonly used to describe it. This model describes a pulsar as a highly magnetised, rotating neutron star, with a dipole magnetic axis that can

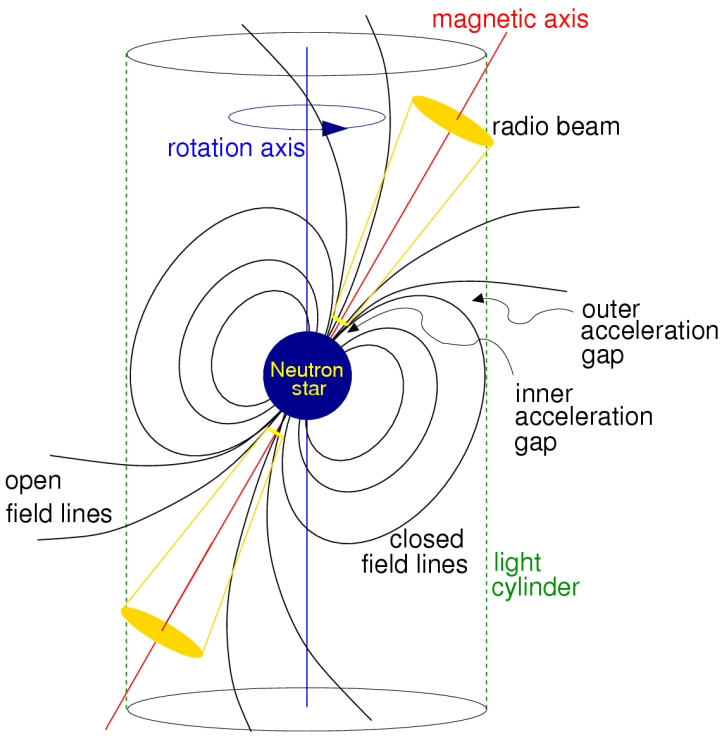


Figure 2.1: A diagram of the traditional magnetic dipole model (lighthouse model) of a pulsar. Figure credit: Lorimer and Kramer [69].

be inclined with respect to the rotational axis. The magnetic field rotates with the neutron star. The further away from the star the field is, the higher its radial velocity. At some point, this radial velocity will be equivalent to the speed of light; the cylinder around the star with this radius is known as the light cylinder. The magnetic field, however, cannot exceed the speed of light. Thus, where the magnetic field lines would extend beyond the light cylinder, the field lines "break" and do not close to meet the other pole. As shown in the figure, this results in two regions above the magnetic poles where the field lines do not close. As a neutron star spins, charged particles are accelerated along its magnetic field lines. The open field lines result in two conical beams of broadband EM emission emanating from the regions above the magnetic poles. If, during the rotational cycle, one or both of the beams pass over the earth, the emission could potentially be detected. Due to the highly stable rotation of the neutron star, this emission is seen as a series of regular pulses, usually in the radio band from $10\,\mathrm{MHz}$ to $10\,\mathrm{GHz}$ typically, and exceptionally up to higher frequencies.

On Earth, pulsars are generally observed as very weak radio sources, to the extent that it is only

possible to detect individual pulses in very few of the most luminous pulsars. In the vast majority of cases, individual pulses are below the level of background noise. Hence, only the largest, most sensitive radio telescopes are able to detect pulsars. The duration of a pulsar observation can range between a few minutes and several hours, and is recorded at high temporal and spectral resolution. These observations cover hundreds to thousands of individual pulses.

In a timing observation, where the spin period of the pulsar is known, this data is *folded* to accurately determine the TOAs of an individual pulse. Folding is a process whereby an observation is separated into sections of length equivalent to the spin period, and these sections are coherently summed together. This summing is intended to increase the signal strength of the pulse relative to background noise. If sufficient pulses are summed, the signal-to-noise ratio (SNR) is increased to a point at which a pulse shape can be determined [46]. This is known as an *integrated pulse profile*. The individual pulses vary greatly in shape and intensity. Thus, if only a small number of pulses are summed, the shape of the pulse profile will vary depending on the specific pulses used to generate it. However, if sufficient pulses are summed together, a stable pulse profile can be generated. These stable profiles vary little over time and can be used to accurately index a position in the pulse. The TOA of an individual pulse can be measured with extreme accuracy – up to 14 significant figures for MSPs [111] – using the known period at which the data is folded and the observed phase of the pulse profile [26, 53].

These highly accurate TOA observations can be used as a powerful measurement tool. A pulsar's spin period is highly stable, thus any observed deviations in the predicted TOAs can be used to measure other factors that may affect the pulse train. The most important of these are gravitational interactions between a pulsar and other bodies, or a range of phenomena that cause distortions in space-time, such as gravitational waves. Most large radio telescopes participate in one or more of the three long-term PTA programs [50, 75, 79]. The aim of these PTAs is to collect long-term high-accuracy TOAs for a number of stable pulsars. These long-term timing programs will enable the detection of long period gravitational waves, such as those generated by the merging of two supermassive black holes.

Only a small fraction (< 3000 [76]) of the 25000 to 120000 potentially observable pulsars [37, 67, 131] have been found, thus there is a continuing drive to find new pulsars. This is not a trivial task and requires extensive observation time and computation time. The majority of

known pulsars have been found through various large-scale surveys [72], which systematically search large contiguous sections of the sky. A smaller number of pulsars have been found with targeted searches, which focus on features such as globular clusters, supernova remnants, and gamma-ray sources.

The pulsar population is divided into two main groups: "normal" pulsars and millisecond pulsars (MSPs). Normal pulsars generally have spin periods of around one second, while MSPs have far shorter spin periods: $P \lesssim 0.03$ seconds. The majority of MSPs are found in a binary system with some other stellar object, and the number of known MSPs is far less than the number of known normal pulsars. Pulsars, especially bright MSPs with narrow pulse profiles, are of great scientific interest as they are highly stable and allow extremely high-accuracy TOA measurements. The various PTAs require many ultra-high precision MSPs, ideally distributed isotropically across the sky. These factors all motivate the many ongoing searches for new, and hopefully exotic, pulsars.

2.1 Pulsar Characteristics

Pulsars can be characterised by a number of key attributes illustrated in the classic "P-$\dot{P}$ diagram" (Figure 2.2). The very large angular momentum of pulsars make them highly stable rotators with exceptionally stable spin periods (P). Spin periods increase very slowly over time in a process known as *spin-down* ($\dot{P}$), which is predominantly attributed to a loss of kinetic energy through magnetic dipole radiation [104]. These two properties form the axes of the P-$\dot{P}$ diagram. If the simplifying assumption of a rotating magnetic dipole is made, P and $\dot{P}$ can be used to infer a number of other basic properties. The surface magnetic field strength (B) can be given as $B \propto (P\dot{P})^{1/2}$. The diagonal dashed green lines in the figure represent constant values of B. In the P-$\dot{P}$ diagram, the magnetic field strength increases towards the top right, where the highly magnetic neutron stars known as magnetars can be found. An age estimate known as the characteristic age $\tau_c = P/(2\dot{P})$ [69] can be derived from P and $\dot{P}$; constant values of τ_c are shown as diagonal dash-dotted lines in the figure. Most of the emissions from a pulsar are generated by accelerating charged particles in the magnetosphere; this is associated with a small reduction in kinetic energy. Thus, the luminosity is proportional to the loss of kinetic energy. This relationship is represented as spin-down luminosity $\dot{E} \propto \dot{P}/P^3$. Accordingly, the young pulsars towards the

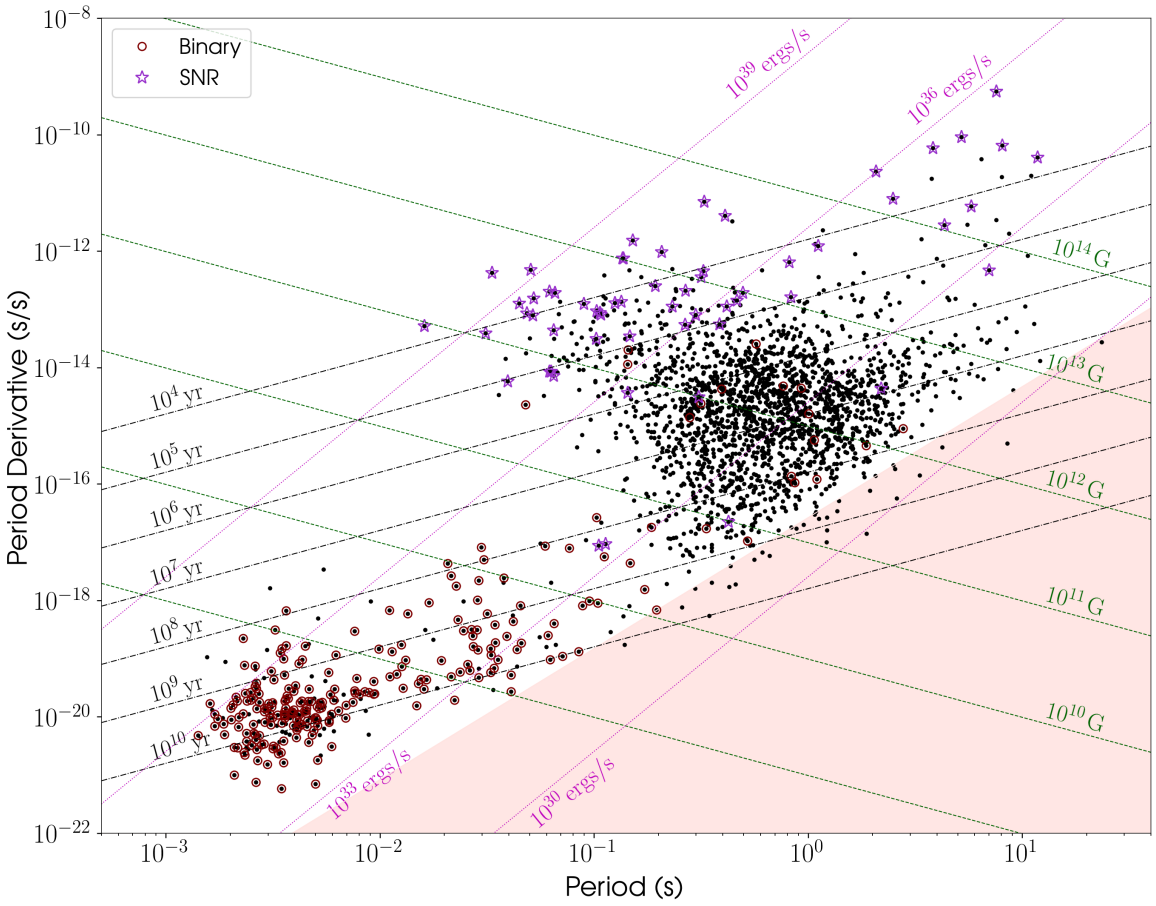


Figure 2.2: This P-$\dot{P}$ diagram shows the current population of known radio pulsars; the circles denote pulsars in binary systems and the stars are pulsars with associated supernova remnants. Lines of constant magnetic field strength B (dashed), characteristic age τ (dash-dotted), and spin-down energy loss rate $\dot{E}$ (dotted) are also shown. The pink shaded region on the right denotes the approximate area where pulsars become too weak to detect.

top left of the figure are the brightest.

The pulsar population is distinctly bi-modal; normal pulsars make up the main cluster, while MSPs are the smaller group in the lower left corner. Normal pulsars generally have pulse periods of around one second, magnetic fields in the order of 10^{12} G, and rotational periods that decrease at a rate of approximately 10^{-15} s/s [68]. On the other hand, MSPs have far shorter spin periods ($P \lesssim 0.03$ seconds), slower spin-downs ($\dot{P} \lesssim 10^{-19}$ s/s), and weaker magnetic fields in the order of 10^{8} G. Most MSPs are in binary systems, this can be attributed to the way MSPs are formed, as discussed in the next section.

In the P-$\dot{P}$ diagram, young pulsars are generally found at the top left of the cluster of normal pul-

sars, and they typically have high luminosity and short spin periods. Some of these young pulsars still have associated supernova remnants, indicated by purple stars in Figure 2.2. Young pulsars quickly (10^{5-6} years) migrate down and right to the main body of normal pulsars. This migration continues for approximately 10^7 years. As the pulsars begin to approach the lower right-hand edge of the main cluster, their luminosity begins to drop. Once their spin-down luminosity drops below 10^{30} ergs/s, they generally become too faint to detect, this is shown by the red shaded area in the figure. The boundary of this region marks what is known as "the death line".

2.2 MSP formation

The MSPs form a separate sub-population and are formed from stars which are in a binary system prior to the formation of the neutron star. This class of pulsars are of particular interest as they are highly stable and often allow the extremely high-accuracy TOA measurements required by PTAs. Furthermore, MSPs in binary orbits are excellent tools for probing and testing our theories of gravity.

In systems where there are two or more gravitationally bound main sequence stars, the most massive of the stars will evolve the fastest. If this star is in the correct mass range it may explode as a supernova and form a neutron star. These supernovae disrupt 90% or more [2] of the systems, however a small number of systems survive as a gravitationally bound pair. In these systems, the neutron star and its companion will continue their normal evolution for a further $\sim 10^{6-10}$ years [68, 74]. If the companion star evolves to become a red giant, it may become so large that it overflows its Roche lobe[1], resulting in its outer atmosphere forming an accretion disk and spiraling in, to fall onto the neutron star. This infalling material decreases the magnetic field and transfers angular momentum to the neutron star, increasing its rotational velocity, and resulting in an MSP. These changes in P and $\dot{P}$ move the MSPs away from the normal pulsars to the bottom left of the P-$\dot{P}$ diagram. This transition is associated with an increase in luminosity, potentially resulting in faint pulsars becoming visible once again, thus MSPs are often referred to as *recycled pulsars*.

The mass of the secondary star determines the type of binary. If the secondary star has a low mass,

[1]The Roche lobe is the distinctively hourglass-shaped region surrounding a star in a binary system within which orbiting material is gravitationally bound to that star

it will evolve into a white dwarf. During the mass-transfer stage of such a low-mass system, the orbits tend to be circularised by tidal interaction in the red giant envelope. Thus, most of the low-mass MSP binary systems have very low eccentricities. If the secondary star has a higher mass, it may continue to form a second neutron star; if the binary system survives the second supernova, the result is a double neutron star binary system. In rare cases, both neutron stars may be observed as pulsars, such as J0737−3039A/B [70]. The formation mechanism of MSPs and their long lifespan explains why over 63% of MSPs are found in some form of orbital system [76], compared to only 1% of normal pulsars (Figure 2.2). This is a somewhat simplified description of the various evolutionary paths, for more detailed descriptions see Lorimer [68] and Stairs [120].

2.3 Finding new pulsars

This section gives an outline of how new pulsars are found, with particular focus on the computationally intensive components of the search pipeline. The aim here is simply to provide the greater context of this work; an actual search pipeline will have many additional steps not covered in this text. For a more in-depth discussion on pulsar search procedures see Lorimer and Kramer [69] and Stovall et al. [124].

The first pulsar was discovered by visual observation of individual pulses on a pen chart [47]. This type of observation of individual pulses is only possible for a small number of the most luminous pulsars. For the vast majority of pulsars, the signal from a single pulse is very weak and dispersed, and is thus generally below the level of background noise. Thus, to find a pulsar, an observation generally has to cover hundreds of thousands of rotations, and the pulsar is detected using the periodic nature of the signal.

The periodic signal from a pulsar can be described parametrically. When searching for pulsars, a number of the parameters (such as sky location, dispersion, and periodicity) have unknown values. The values of these *a-priori* unknown parameters (referred to hereafter as *search parameters*) are determined by iterating over trial values for each parameter and performing some operation on the observed signal. For the location parameter these iterations consist of a number of observations, while the values of the remaining search parameters dictate some DSP operation; these observations and DSP tasks are the components of a pulsar search pipeline. Thus, the number and range of trial values of the search parameters are the most significant factors determining the

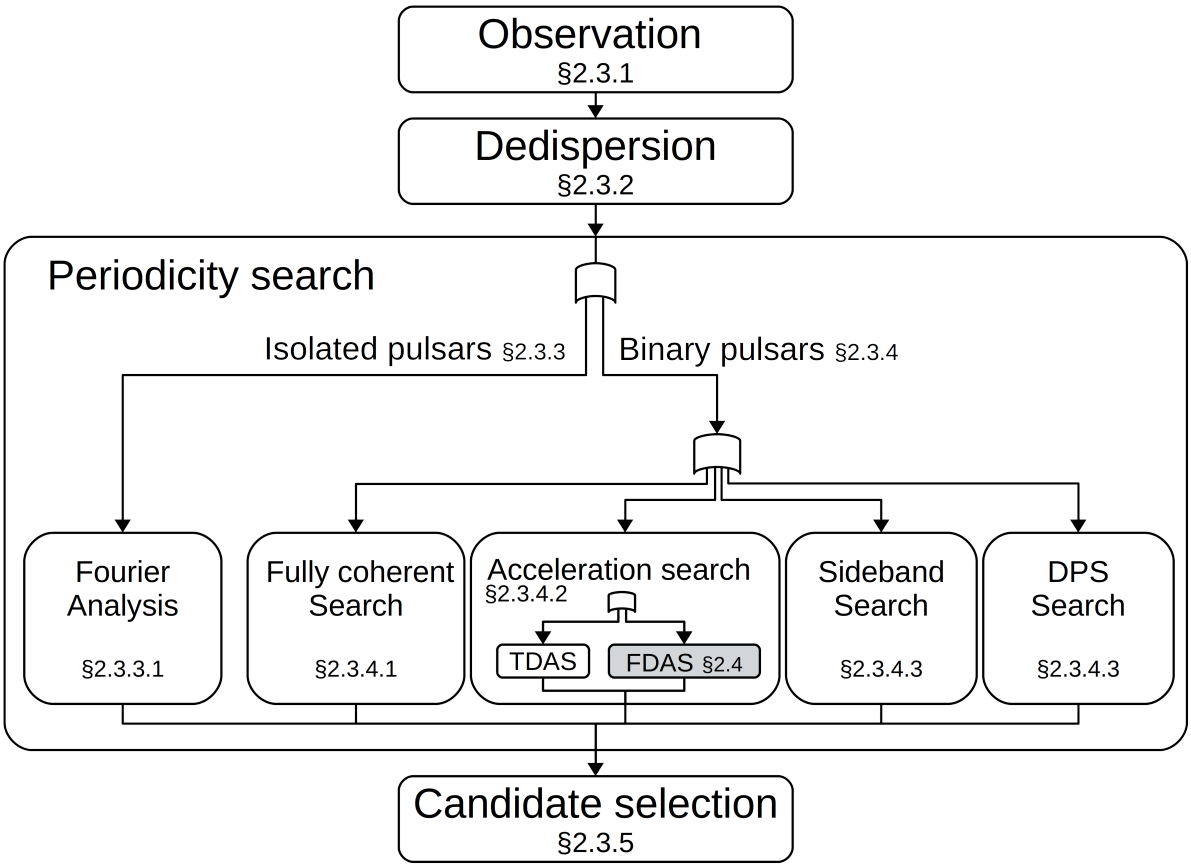


Figure 2.3: The main components of the pulsar search pipeline. This work focuses on the FDAS, which is one of the primary means of performing the periodicity search.

computational requirements of a search.

There are two types of search, *targeted* and *blind* searches. In a targeted search, the range of some of the search parameters are constrained. For instance, when searching for pulsars in a globular cluster the location and dispersion may be known, in which case as little as one observation or *pointing* is required. In a blind search, the values of the search parameters are unknown, thus the search has to cover a much wider range of the parameter space.

This section describes the four main components of the search pipeline: observation, dedispersion, periodicity search, and candidate selection. These components are shown in Figure 2.3 and each is discussed in detail below. The primary component that differs between the searches for binary and isolated pulsars is the periodicity search. Most pulsars in long period binaries can be detected with the standard periodicity search (Section 2.3.3). However, when the orbital period is short, this component needs to be expanded to cover the additional parameters required to describe the orbital motion of a binary pulsar. The increased number of search parameters

makes this component particularly computationally intensive, to the extent that it is unfeasible to perform a periodicity search that covers the full range of the parameter space. As such, there are a number of alternative approaches to performing a periodicity search (Section 2.3.4), each of which searches a specific sub-range of the parameter space. The FDAS (Section 2.3.4.2) is a comparatively efficient periodicity search that covers the section of search parameter space in which the majority of known binary pulsars are found. Our work is a highly efficient GPU-accelerated implementation of the FDAS.

2.3.1 Observation

The first parameter covered in a search for pulsars is its location in the sky, this parameter is determined by the observation component. A targeted search may consist of as little as one observation (or pointing); the duration of targeted observations can range from a few minutes up to several hours. However, a large-scale blind survey may consist of thousands of pointings, and the duration of individual observations is generally shorter: 5 to 15 minutes each [72].

Search observations are generally performed with very large radio telescopes, most of which are some form of parabolic reflector which focuses incoming radio waves onto a feed horn. The feed horn is usually a broadband ($30\,\mathrm{MHz}$ to $850\,\mathrm{MHz}$ [10, 72]) antenna which converts EM radiation into an analogue voltage. This complex voltage is usually captured in two orthogonal polarisations and is then amplified by a receiver. The raw signal is then mixed with a low-frequency signal and passed through a band-pass filter, producing the intermediate frequency feed which is transmitted to a receiver backend.

In these types of radio telescopes, the size of telescope is large compared to the wavelength being observed. This means that they will be most sensitive to radio waves coming from a small area of the sky, in what is known as the "main beam" of the telescope. In an effort to decrease the number of pointings, most contemporary large-scale pulsar surveys have multiple beams per pointing, allowing them to simultaneously observe multiple sky locations. In single dish telescope, this can be achieved by using a focal plane array (FPA), where an array of receivers is placed at the focus of the telescope. The very fruitful Parkes multibeam pulsar survey (PMPS) used a 13 element FPA, allowing each pointing to generate 13 beams. In multi-dish telescopes, signals from separate antennas can be combined digitally to form beams, in a process known as tied-array beamforming

[13]. Each beam is associated with a specific sky location, making this the associated with the location of a pulsar. In the initial pulsar searches that are proposed for the SKA1-Low and SKA1-Mid telescopes, each pointing will have 500 and 1500 tied-array beams respectively [58, 66]. The beamforming required for these searches represents a substantial computational requirement [66].

In the receiver backend, the signals from each beam will usually be split into independent narrow frequency channels, usually between 128 and 2048 channels per beam [72]. The signal in each channel is digitised at high temporal resolution ($50\,\mu s$ to $100\,\mu s$), and each sample is recorded with one to eight bits. This channelised digital signal is passed down the pipeline for further processing, starting with dedispersion.

2.3.2 Dedispersion

As a pulse propagates through the ISM, the EM waves interact with ionised particles causing a frequency-dependent delay. This disperses the pulse, so that it is observed at higher frequencies before lower frequencies. The delay is inversely related to the square of the radio frequency, and the degree of the dispersion is quantified by the term dispersion measure (DM). The dispersion needs to be corrected through a process known as dedispersion before the signal can be searched for periodic signals. Formally, the observed time delay, Δt, between two frequencies v_1 and v_2 is given by:

$$\Delta t = k_{\mathrm{DM}}\,\mathrm{DM}\left(v_1^{-2} + v_2^{-2}\right) \tag{2.1}$$

where $k_{\mathrm{DM}} = 4.148808 \times 10^3\ \mathrm{MHz}^2\ \mathrm{pc}^{-1}\ \mathrm{cm}^3\ \mathrm{s}$ is the dispersion constant [78], the frequencies v_1 and v_2 are in MHz and the DM is given in $\mathrm{pc\,cm}^{-3}$.

If the DM is known, as is the case when timing known pulsars, the signal can be fully corrected with *coherent dedispersion*[2] [45], a process that works directly on the complex voltages prior to digitisation. However, if the dispersion is *a-priori* unknown, the observation must be dedispersed for a number of trial DM values. A modern blind survey may cover up to several thousand DM trials [72], making DM one of the primary search parameters. Due to the computational intensity of coherent dedispersion, it is not feasible to perform this process for a large number of trial DM values. In these cases, *incoherent dedispersion* [125] is used. There are three methods

[2]Here coherent means that the wavelength, phase and amplitude of a periodic signal are preserved.

of performing incoherent dedispersion; the simplest, known as *direct incoherent dedispersion* is described here and the other two are discussed in Section 4.6.1. Direct incoherent dedispersion operates on the channelised data, applying a per-channel, DM-dependent time delay. Once all the channels have been shifted in time, they can be summed across frequency to form a single dedispersed time series. Thus, the output of the dedispersion component is such a time series for each trial DM.

The DM step size and the width of the channels are chosen so that the dispersion within a channel is negligible. There will, however, always be some dispersion delay across a frequency band; this has the effect of "smearing" the signal in the band. The width, and thus the number of channels, is chosen so as to keep the error introduced by this smearing below some desired level. Therefore, the dedispersion requirements dictate the spectral resolution (the number of frequency channels) in pulsar search data. Searches that cover shorter spin periods, such as those for MSPs, require higher-frequency resolution. Search data is generally recorded with hundreds to thousands of channels, with the number of channels usually chosen to be a power of two.

2.3.3 Periodicity searches: isolated pulsars

The next step in the pipeline is to search each dedispersed time series for periodic signals. The periodicity of the signal will reveal the spin frequency of the pulsar. Detecting a periodic signal, such as that of a isolated pulsar, is typically performed with Fourier analysis. Fourier analysis is an extremely powerful and widely used tool, upon which this work is heavily based. A robust understanding of this topic will aid the reader in engaging with this work. However, only a brief outline of Fourier analysis is given here; for a more thorough treatment, the reader is referred to the wealth of literature on the topic, with good pulsar-specific reviews found in Johnston and Kulkarni, Lorimer and Kramer, Ransom et al. [51, 69, 115] and Groth [43].

It is worth noting that there is an alternative periodicity search that makes use of the fast folding algorithm (FFA) [119]. Due to its comparative computational inefficiency, this method has rarely been used in pulsar searches, however, there has been renewed interest in it in some recent surveys [19, 105]. This work focuses on periodicity searches based on the far more commonly used Fourier techniques.

2.3.3.1 The discrete Fourier transform

The fundamental principal of Fourier analysis is that any periodic signal can be expressed as the sum of a number of pure sinusoidal signals. Each of these sinusoidal signals has a unique wavelength and some phase and amplitude. Given a dedispersed, uniformly sampled time series n_j ($j = 0, 1, 2, ..., N_t - 1$), the DFT of the N_t samples produces a set of N_t complex Fourier components, with the k^{th} Fourier component given as:

$$\mathcal{F}_k = \sum_{j=0}^{N_t-1} n_j e^{-i2\pi jk/N_t} \tag{2.2}$$

where $i = \sqrt{-1}$ and k is the *Fourier frequency* ($k = 0, 1, ..., N_t - 1$). If the time samples are separated by Δt seconds, the duration of the observation (T_{obs}) is given as $T_{\text{obs}} = \Delta t N_t$ and the frequency of the k^{th} Fourier component is $f_k = k/(N_t \Delta t) = k/T_{\text{obs}}$. Frequency separation is given as $\Delta f = 1/T_{\text{obs}}$; thus, the components are referred to as "Fourier bins", and the width of these is Δf. This gives the finest frequency resolution available, while maintaining completely independent Fourier components. The frequency $N_t/(2T_{\text{obs}})$ is known as the Nyquist frequency. In cases such as ours, where the input is real-valued, the DFT is symmetric about the Nyquist frequency such that $\mathcal{F}_{N_t-k} = \mathcal{F}_k^*$, where $\mathcal{F}_k^*$ is the complex conjugate of $\mathcal{F}_k$. This means that all the information of the real-to-complex DFT can be represented with $N_t/2$ complex Fourier components.

Each Fourier component describes one of the sinusoidal components of the original signal. The Fourier frequency gives the wavelength, and the angle of the complex value describes the phase. In addition, the magnitude of the complex value ($\mathcal{A}_k = |\mathcal{F}_k|$) describes the amplitude of the wave. A power spectrum (Figure 2.4b) can be generated by calculating the powers ($\mathcal{P}_k = |\mathcal{F}_k|^2$) of the Fourier components. A large power indicates a large amplitude and thus a strong periodic component of the original signal. Thus, finding a significant value in a power spectrum is the fundamental means of detecting a periodic signal. In a periodicity search, powers above some statistically determined threshold are considered pulsar candidates. These pulsar candidates are the input to the final component of the search pipeline (Section 2.3.5). This type of analysis will reveal the spin frequency of a pulsar and has proven to be a very powerful tool in the detection of isolated pulsars.

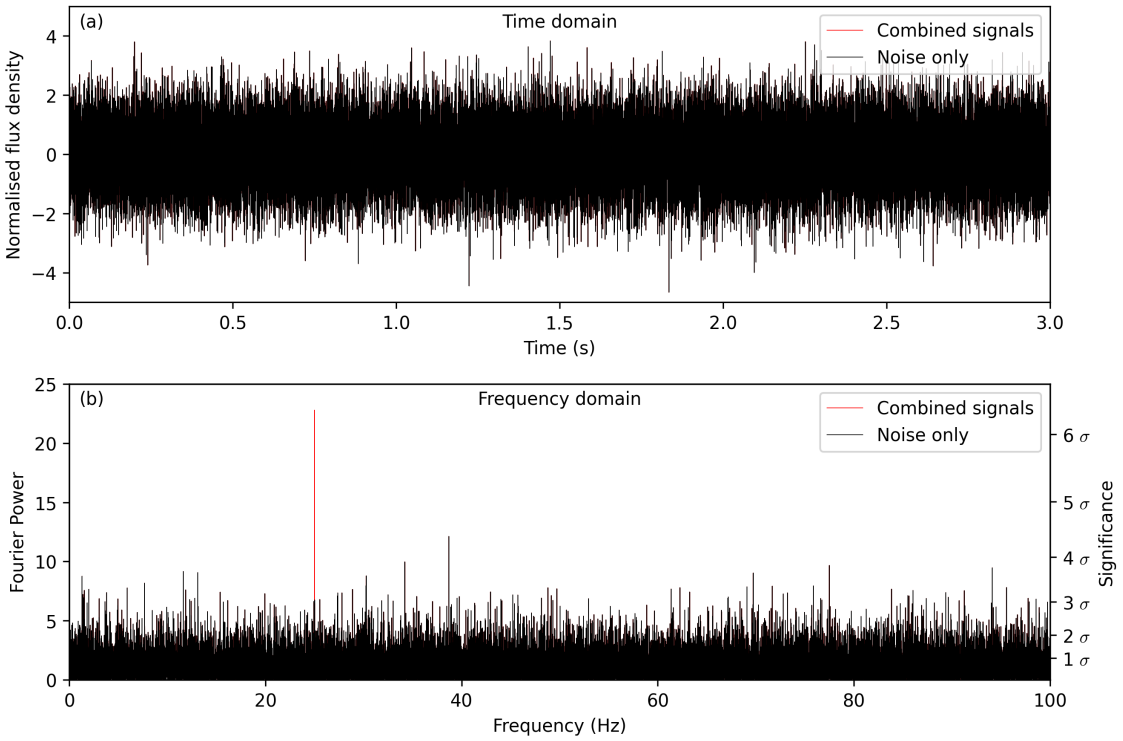


Figure 2.4: A power spectrum.
(a) Plotted in black is a time series of normally distributed noise. Below this, in red, is the same noise combined with a $25\,\mathrm{Hz}$ sinusoidal signal of amplitude 0.0035. Due to the very small amplitude of this signal, the two are practically indistinguishable in the time domain.
(b) The power spectrum of the DFT of the combined signals is shown in red. As above, the power spectrum of only the noise is plotted over this in black. Here, the weak sinusoidal signal clearly makes a discernible peak in the power spectrum of the combined signals.

This type of transform converts a signal from a time series in the time domain to a spectrum in the frequency domain. Figure 2.4 shows that in the time domain, the "information" of a periodic signal is distributed across the entire time series, while in the frequency domain, the same information is localised into a small number of Fourier components. Each Fourier power can be considered a trial value for the f_{spin} . Searching through a power spectrum is a computationally trivial task, however, performing the Fourier transform (FT) to generate these values is not. The computational complexity of the DFT is $\mathcal{O}\left({N_t}^2\right)$. However, when N_t is a power of two, the computationally efficient FFT can be used which has a complexity of $\mathcal{O}\left(N_t \log_2(N_t)\right)$. Thus, most observations are specifically chosen to have a number of samples that is some large power of two.

2.3.3.2 Increasing frequency resolution

One of the main limitations of the DFT is that its frequency response is only ideal for signals that exactly match the centre frequencies of the Fourier bins. The loss of sensitivity for frequencies that fall outside the Fourier bin centres is known as "scalloping loss" [82] and can be as high as 36.3% for frequencies midway between two adjacent bins [115]. The three main methods used to mediate this loss are zero padding, Fourier interpolation and interbinning. Zero padding is the simplest of these procedures, and involves simply adding a large number of points – with value equivalent to the mean of the data[3] – to the end of the time series. This artificially increases the value of N_t and thus $T_{\rm obs}$, which in turn increases the Fourier resolution. This technique is simple, however, increasing the length of the time series can significantly increase the computational requirements of a search, as computing long FFTs can be very computationally intensive.

Fourier interpolation is a tool that can be used to calculate a complex Fourier component at an arbitrary frequency $f_r = r/T_{\rm obs}$, where r is any real number. This technique is based on the premise that a fully coherent response for virtually any signal can be completely recovered with a matched filter in the frequency domain. Increasing the frequency resolution in this manner is referred to as *fine binning*. Full details of Fourier interpolation can be found in Ransom et al. [115]. In summary, the complex Fourier component at any real-valued frequency r can be given as:

$$\mathcal{F}_r = \sum_{k=0}^{k=N-1} \mathcal{F}_k e^{-i\pi(r-k)} \ \mathrm{sinc}\left[\pi(r-k)\right] \tag{2.3}$$

The efficiency of Fourier interpolation comes from the fact that $\mathrm{sinc}\left(\pi(r-k)\right) \to 0$ as $\left(\pi(r-k)\right) \to \pm\infty$ meaning the expansion of $\mathcal{F}_r$ is dominated by the $\mathcal{F}_k$ in close proximity to r. Thus,

$$\mathcal{F}_r \simeq \sum_{k=[r]-\frac{m}{2}}^{k=[r]+\frac{m}{2}} \mathcal{F}_k e^{-i\pi(r-k)} \ \mathrm{sinc}\left[\pi(r-k)\right] \tag{2.4}$$

where m is the number of neighbouring Fourier bins used in the interpolation and $[r]$ is the closest

[3] If a signal is not normalised, padding with zeroes can add low-frequency power to the spectrum, whereas using the mean adds power to only the first spectral bin.

integer to r. This allows one to calculate an approximate coherent Fourier component at any real-valued Fourier frequency r from the surrounding m bins. Fourier interpolation is a correlation of the m DFT components surrounding the frequency of interest with a "template" response. This method is advantageous in that it yields a close-to-coherent result at any frequency below the Nyquist frequency, for relatively small values of m ($m \simeq 32$). The correlation can be calculated with an appropriate finite impulse response (FIR) filter, the coefficients of which are given in Equation 2.5. These coefficients are given in terms of their distance from r ($\Delta r = r - k$).

$$I_{\Delta r} = e^{-i\pi(\Delta r)} \operatorname{sinc}\left[\pi(\Delta r)\right] \tag{2.5}$$

A less computationally expensive method known as "interbinning" is widely used in the Fourier analysis of pulsar data [69, 82, 115]. This method approximates Fourier components at half-integer frequencies using only the two neighbouring integer frequency bins, and is given by Equation 2.6.

$$\mathcal{F}_{1+\frac{1}{2}} \simeq \pi/4\left(\mathcal{F}_k - \mathcal{F}_{k+1}\right) \tag{2.6}$$

This technique reduces the maximum loss of SNR from $\sim$36.3% at frequency offset of half a bin to $\sim$7.4% at an offset of $\pm\left(1 - \frac{\pi}{4}\right)$ bins. This represents a significant reduction in the loss of sensitivity due to scalloping, for a minimal amount of computation. However, interbins as defined in Equation 2.6 differ from regular integer DFT bins or Fourier interpolated components in three ways. Firstly, the phase of the interbins is not correct. Secondly, each interbin is correlated with the integer bins it was created from, meaning that interbins are not independent Fourier trials. Lastly, they have different noise properties, and consequently, calculating the significance of interbin powers is much more difficult. Despite these drawbacks, interbins are still useful in a periodicity search, as they are only used to identify candidate signals. Then, the true properties and significance of the identified candidate can be determined using a large-scale Fourier interpolation.

2.3.3.3 Increasing sensitivity to narrow pulses

The FT is very effective at detecting periodic signals, specifically sinusoids. However, many periodic signals such as those of pulsars are not sinusoidal, they often have a small duty cycle

(D) [76]. The duty cycle of a periodic signal describes how narrow the periodic signal is, in the case of pulsars, D is the pulse width (PW) over spin period. A low duty cycle signal can be created by summing a number of sinusoidal components, each with a wavelength which is an integer fraction $(1/\mathbb{Z}_+)$ of that of the original periodic signal. These sinusoidal components represent the harmonics of the periodic signal; the first is called the *fundamental* harmonic, and has the same wavelength as the signal. A sinusoidal signal will have only one harmonic, and the number of significant harmonics will increase as the duty cycle decreases. Most radio pulsars have duty cycles below 5%, corresponding to $\gtrsim 10$ significant harmonics [115]. The power spectrum of such a signal (Figure 2.5a) will contain many significant values, as the "power" of the signal is spread across the harmonics of the signal. In a power spectrum, the harmonics are found at integer multiples of the fundamental frequency. This spreading of the power decreases the power of the fundamental bin by approximately D^2 [130]. This decreases a search's sensitivity to signals with low duty cycles.

A technique known as *incoherent harmonic summing*, described by Taylor and Huguenin [126], can be used to increase the sensitivity of a frequency domain search by summing harmonically related powers. This is done by summing the first h harmonically related powers to the fundamental. Summing powers discards the phase information of the Fourier components and is thus considered incoherent. This type of incoherent summing can, however, increase the sensitivity of a search by a factor of $\sqrt{h}$ [115]. Thus, harmonic summing can be used to increase a search's sensitivity to periodic signals with a low duty cycle. As seen in Figure 2.5a, the power in each of the separate harmonics of the signal is not significantly greater than that of the noise. However, after summing 8 harmonics, the power of the periodic signal is gathered into a number of bins which are significantly above the level of the noise. Most frequency domain pulsar search algorithms inspect the original spectrum and harmonically summed versions, which contain the sum of the first 2, 4, 8, and 16 harmonics [69]. Thus, the number of harmonics summed (h) can be considered to be one of the search parameters. However, as it has only 5 trial values, it is regarded as one of the least significant search parameters. Thus the search parameters of the "classic" search pipeline for isolated pulsars are: location, DM, f_{spin}, and h.

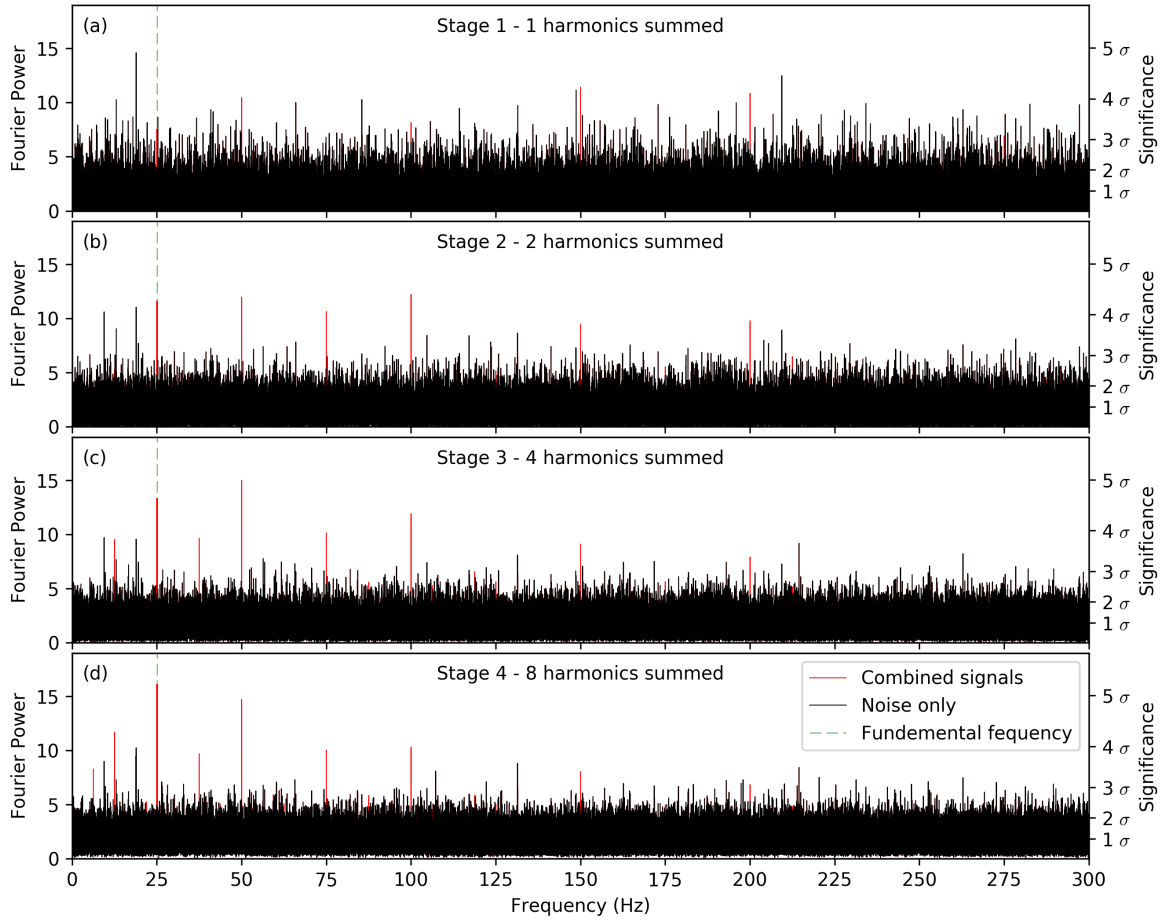


Figure 2.5: Incoherent harmonic summing.
(a) Plotted in red is the normalised power spectrum of normally distributed noise, combined with a low duty cycle periodic signal of 25 Hz and amplitude 0.00085. Superimposed over this in black is the power spectrum of only the noise. The harmonics of a pulsar's periodic signal appear as a series of evenly spaced peaks in a power spectrum.
(b-d) The normalised power spectrum after 2 - 4 stages of incoherent harmonic summing. This process can recover some of the power spread across the harmonics.

2.3.4 Periodicity searches: binary pulsars

Detecting the periodic signal from a pulsar in a binary system can be significantly more challenging than finding the periodic signal of an otherwise-similar isolated pulsar. The orbital motion Doppler shifts the arrival times of the pulses, thus the observed spin frequency appears to change through the orbit. When an observation containing a signal with this type of modulated frequency is Fourier transformed, the power (of each harmonic) is "smeared" over a number of spectral bins, as is shown in Figure 2.6. This type of smearing (not to be confused with smearing due to dispersion) reduces the SNR in the individual spectral bins. Hence, the classic periodicity search

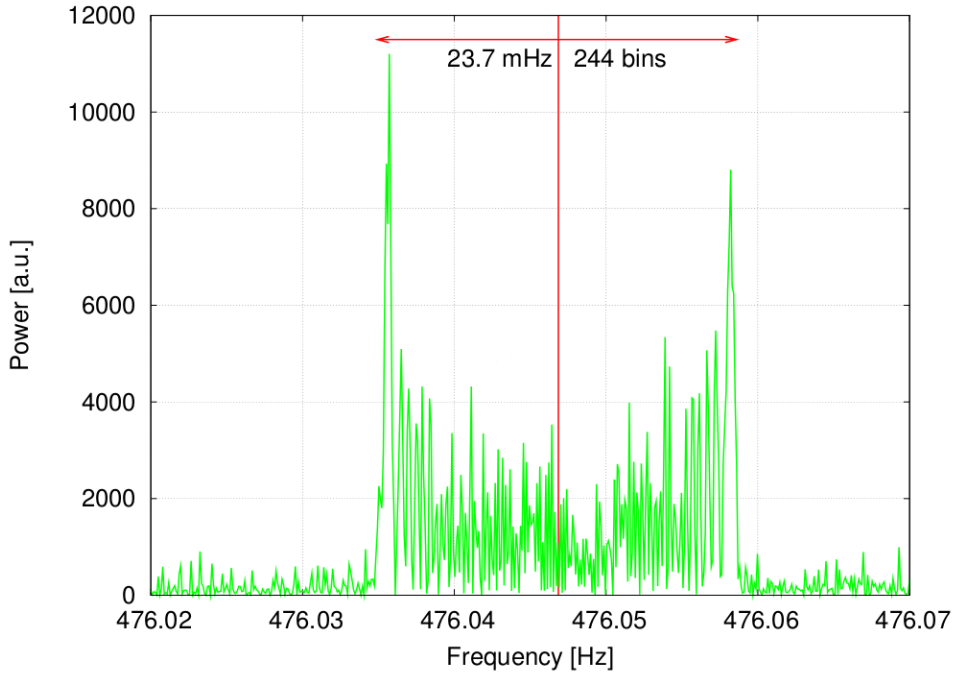


Figure 2.6: Power spectrum showing the spread of power over Fourier bins. This power spectrum is taken from a 2.9 hour time series recorded at the Parkes Observatory, Australia. The pulsar's intrinsic frequency is shown as the line at $476.0469\,\mathrm{Hz}$. Since the pulsar is in a circular binary orbit, the signal power is distributed over a wide range of 244 frequency bins or $23.67\,\mathrm{mHz}$. (Image taken from Aulbert [6])

is less sensitive to pulsars in binary systems.

A number of techniques have been developed to compensate for this loss in sensitivity. It is possible to completely recover this sensitivity with a *fully coherent* search, however this requires a very large parameter space to be surveyed and is thus highly computationally intensive. A number of additional search techniques have been developed to search various sub-sections of the full parameter space. These methods trade some loss in sensitivity for computational efficiency and can be divided into three main classes: *acceleration* searches, *dynamic power spectrum (DPS)* searches, and *sideband* searches. The metric used to separate these groups is the ratio of T_{obs} to orbital period (P_{orb}). When the observation covers only a small section of the orbit $(T_{\mathrm{obs}} \ll P_{\mathrm{orb}})$, an acceleration search can be used. When the observation is a similar length to the orbit $(P_{\mathrm{orb}} \simeq T_{\mathrm{obs}})$, some form of DPS search is more sensitive, while the sideband search is effective when the observation is longer than the orbital period $(T_{\mathrm{obs}} \gg P_{\mathrm{orb}})$. Each of these three search types is detailed in the sections that follow. The majority of known binary pulsars fall into the range where an acceleration search is effective, making this the most widely used alternative. There

are two main variants of the acceleration search, the time domain acceleration search (TDAS) [20, 51] and the frequency domain acceleration search (FDAS) [115].

2.3.4.1 Fully coherent searches

It is possible to completely recover the lost sensitivity by resampling the dedispersed time domain data to compensate for the Doppler shift [51]. This resampling transforms the time series from the observation time frame (t) to the pulsar time frame (τ). In the pulsar time frame, the pulses are returned to their highly regular state, allowing the fully coherent signal to be recovered. The resampled data can be searched with the classic periodicity search (Section 2.3.3) and will completely recover the power lost due to the orbital motion.

The transformation from t to τ is given as:

$$\tau(t) = \tau_0 \left(1 + v(t)/c\right) \tag{2.7}$$

where $v(t)$ is the radial velocity at time t, c is the speed of light and τ_0 is a normalisation factor. For pulsar searching, the Keplerian model is sufficient to describe the orbital motion of most non-relativistic orbits [29]. Using this model, $v(t)$ can be expressed using five independent orbital parameters[4] [29].

When searching for pulsars, the orbital parameters are *a-priori* unknown. Thus, a fully coherent search making use of this transform would have to span a nine-dimensional parameter space: location, DM, the five orbital parameters, f_{spin}, and h. These parameters map to the components of a fully coherent search, the procedure of which is outlined below. The process begins with a number of observations, each of which generate a channelised time series for each pointing. These are all dedispersed for a range of DM trials, and the resulting time series are resampled for every combination of the five orbital parameters. All of the resulting resampled time series are Fourier transformed, and harmonically summed. The resulting power spectra are then searched for significant powers in the same manner as a classic periodicity search.

Performing a fully coherent search is considered intractable. Consider the example of a fully

[4] In the Keplerian model, orbital position is expressed using the orbital period and the standard six Keplerian parameters. However, the representation of $v(t)$ can be simplified to five parameters by combining the semi-major axis (a) and inclination (i) into one parameter and discarding the *longitude of the ascending node* (Ω).

coherent search covering 500 DM trials of all 1500 beams of a single nine-minute SKA1-mid pointing [66], in which each of the Keplerian parameters is searched with 100 trial values. This would require approximately 600 000 years of compute time with a single contemporary desktop GPU or a 64-core CPU. Thus, performing a large-scale fully coherent search covering the full range of all nine parameters is not remotely feasible (in terms of both time and cost) with current computational technology [29]. Hence, a number of alternative methods have been developed, of which acceleration searches are the most widely used. Each of these alternatives targets a specific subset of the full parameter space.

2.3.4.2 Acceleration searches

Acceleration searches are the most successful and commonly used class of search for binary pulsars. This type of search can be used when the length of the observation is significantly shorter than the orbital period. In these cases, the observed change in radial velocity appears to be close to linear. A linear change in velocity implies a constant acceleration, thus $v(t)$ can be expressed with a single "acceleration" parameter (a):

$$v(t) = at \tag{2.8}$$

A pulsar with a fixed line-of-sight acceleration results in a constant rate of change in the observed spin frequency. There are a number of techniques by which a constant frequency derivative can be compensated for, to recover a coherent signal. Thus, in an acceleration search, the parameter space being searched reduces to five parameters: sky location, DM, a, f_{spin} and h – a far more computationally tractable parameter space.

The true line-of-sight acceleration is only close to constant for $T_{\text{obs}} \lesssim 0.1 \times P_{\text{orb}}$ [87, 114], limiting the range in which acceleration searches are effective. However, most large-scale pulsar surveys fall well within this range as T_{obs} is usually in the order of 1 to 15 minutes, while most known binary pulsars have orbital periods greater than 100 minutes[5] [76]. Thus, acceleration searches have proven very successful and have been used to find the vast majority of pulsars in binary systems. There are two main variants of the acceleration search, the time domain acceler-

[5]There may actually be some selection bias here, as acceleration searches are the main tool used to find pulsars in binary orbits, limiting the capability of finding very short period binaries.

ation search (TDAS) [20, 51] and the FDAS [115].

Time domain acceleration search

An acceleration search can be performed in the time domain by resampling a dedispersed time series and then performing a standard periodicity search, as described in Section 2.3.4. A fully coherent search requires resampling a time series for every combination of the five orbital parameters. However, if the line-of-sight acceleration is assumed to be constant, the radial velocity can be expressed with just one parameter (Equation 2.8). This drastically reduces the complexity of the time domain resampling described by Equation 2.7.

The TDAS has been used in many of the large-scale pulsar surveys, and applied to much of the data collected at the Parkes radio telescope. Initially, Camilo et al. [20] used this method to perform a number of targeted searches of the globular cluster 47 Tucanae. These searches discovered nine new MSPs — all in binary systems. The original observations were up to 4.6 hours long (2^{27} samples). However, partly due to computational limitations [20], the acceleration searches were performed on contiguous blocks of 17.5 minutes (2^{23} samples) of data, each of which was searched with 200 acceleration trials. The PMPS has been the most fruitful survey to date, resulting in the discovery of over 800 pulsars [76]. In the initial processing of the data an incoherent TDAS was performed, as at the time, performing a coherent TDAS was not possible with the available computational resources [38]. However, Eatough et al. reprocessed the PMPS data performing a coherent TDAS using The University of Manchester's TIER2 computing facility [33]. This reprocessing performed 59 acceleration trials per DM and found 16 new pulsars, only one of which was an MSP. The High Time Resolution Universe (HTRU) survey is an all-sky survey for pulsars and radio transients, observations are performed with a high time resolution ($64\,\mu s$) and a $340\,\mathrm{MHz}$ observing band centred at $1352\,\mathrm{MHz}$ [56]. This survey divides the sky into three regions: low, mid and high latitudes. Most pulsars are found at low latitudes, thus the T_{obs} for the low-latitude observations is 4300 seconds, significantly longer than most previous large-scale surveys. As part of the data-processing pipeline, these low-latitude observations are searched using a TDAS [87]. Due to the long integration time, each observation is searched using a "partially coherent segmented acceleration" search, a technique whereby each observation is segmented into one, two, four and eight sections. Each of these is searched using the TDAS, increasing the search's sensitivity to pulsars with short orbital periods. Thus far, the HTRU survey

has discovered 32 new binary pulsars [76].

The TDAS has seen widespread use in large-scale pulsar surveys. However, when using the TDAS, each dedispersed time series is resampled for a range of trial accelerations, each resultant time series is then Fourier transformed and searched in the standard manner. This method requires the FT of a full-length data set for each combination of the location, DM and acceleration parameters. Due to the ordering of the search components, this method has a higher computational complexity than its frequency domain counterpart, which has been less frequently used in large-scale pulsar surveys.

Frequency domain acceleration search

An acceleration search can be performed in the frequency domain using the *correlation method*. This variant of the periodic search is the focus of our work. This section gives an overview of the theoretical basis of the correlation method, the full details of which can be found in Ransom et al. [115]. A detailed examination of the existing serial CPU implementation of this method is given in Chapter 3.

The FDAS uses a frequency domain matched filter to compensate for a periodic signal that exhibits a constant change in frequency, such as a pulsar with a constant line-of-sight acceleration. In the frequency domain, a constant acceleration has the effect of "smearing" the coherent sinc-like[6] Fourier response of a periodic signal across a number of Fourier bins, as shown in Figure 2.6. The aim of the correlation method is to recover the coherent, acceleration-corrected Fourier response from the smeared spectrum.

The smearing can be thought of as the convolution of the coherent sinc-like response, with some acceleration-specific impulse response filter. The tails of this filter are infinite but quickly tend towards zero. This means that the majority of the power is smeared across a number of bins in close proximity to the "central" frequency of the signal. If it is assumed that the majority of the power is spread across m bins, one can consider the filter to be a FIR filter of length m bins. If the form and phase of this FIR filter is known, the coherent response can be recovered by correlating the observed smeared Fourier response with the frequency reversed and complex conjugated template that matches this FIR filter. A detailed derivation of the filter can be found

[6] Resembling the sinc function.

in Ransom et al. [115].

A pulsar with constant line-of-sight acceleration will cause a linear shift in the observed spin frequency. Thus, in the frequency domain, acceleration can be expressed as the number of Fourier bins by which the frequency changes during the observation ($\dot{r}$) with $\dot{f} = \dot{r}/{T_{\text{obs}}}^2$. Our work operates in the frequency domain, thus, in this text, frequency will usually be expressed as r and acceleration as $\dot{r}$. For a periodic signal that has a linear change in frequency due to constant line-of-sight acceleration, with mean frequency r and a change in frequency of $\dot{r}$, the coherent, acceleration-corrected Fourier response is given by Equation 2.9 [115, Equation 39]:

$$\mathcal{F}_{r,\dot{r}} = \sum_{k=[r]-m/2}^{k=[r]+m/2} \mathcal{F}_k \frac{1}{\sqrt{2\dot{r}}} \left(e^{i\pi q_k^2/\dot{r}} \left[(\mathcal{S}(Z_k) - \mathcal{S}(Y_k)) - i\left(\mathcal{C}(Y_k) - \mathcal{C}(Z_k)\right) \right] \right) \tag{2.9}$$

The distance of the k^{th} bin to r is given as $\Delta r_k = (r - k)$. The other terms in Equation 2.9 are: $q_k = \Delta r_k - \dot{r}/2$, $Y_k = \sqrt{2/\dot{r}}\,[\Delta r_k - \dot{r}/2]$, $Z_k = \sqrt{2/\dot{r}}\,[\Delta r_k + \dot{r}/2]$ and the functions $\mathcal{C}$ and $\mathcal{S}$ are the normalised Fresnel integrals, which are defined as:

$$\begin{aligned} \mathcal{C}(x) &= \int_0^x \cos\left(\frac{\pi}{2}t^2\right) \mathrm{d}t \\ \mathcal{S}(x) &= \int_0^x \sin\left(\frac{\pi}{2}t^2\right) \mathrm{d}t \end{aligned} \tag{2.10}$$

In Equation 2.9, the acceleration-corrected Fourier response ($\mathcal{F}_{r,\dot{r}}$) is the sum of the products of a number of evenly spaced DFT bins ($\mathcal{F}_k$) with some term. This term can be considered a function of $\dot{r}$ and Δr. Thus, the sum can be thought of as a convolution of an *acceleration filter* – represented by the term – with the observed Fourier components. This convolution is essentially a correlation between the observed DFT and an acceleration FIR filter, hence the name of the method. This method allows the calculation of the acceleration-corrected Fourier response at any combination of frequency and acceleration.

The coefficients of this acceleration filter are a function of $\dot{r}$ and Δr, and the magnitudes of these coefficients are shown in Figure 2.7. Calculating the corrected response for a specific r and $\dot{r}$ can be thought of as a convolution of the DFT bins centred on r and the corresponding horizontal slice from Figure 2.7. It can be seen that the magnitude of the filter coefficients outside the lines

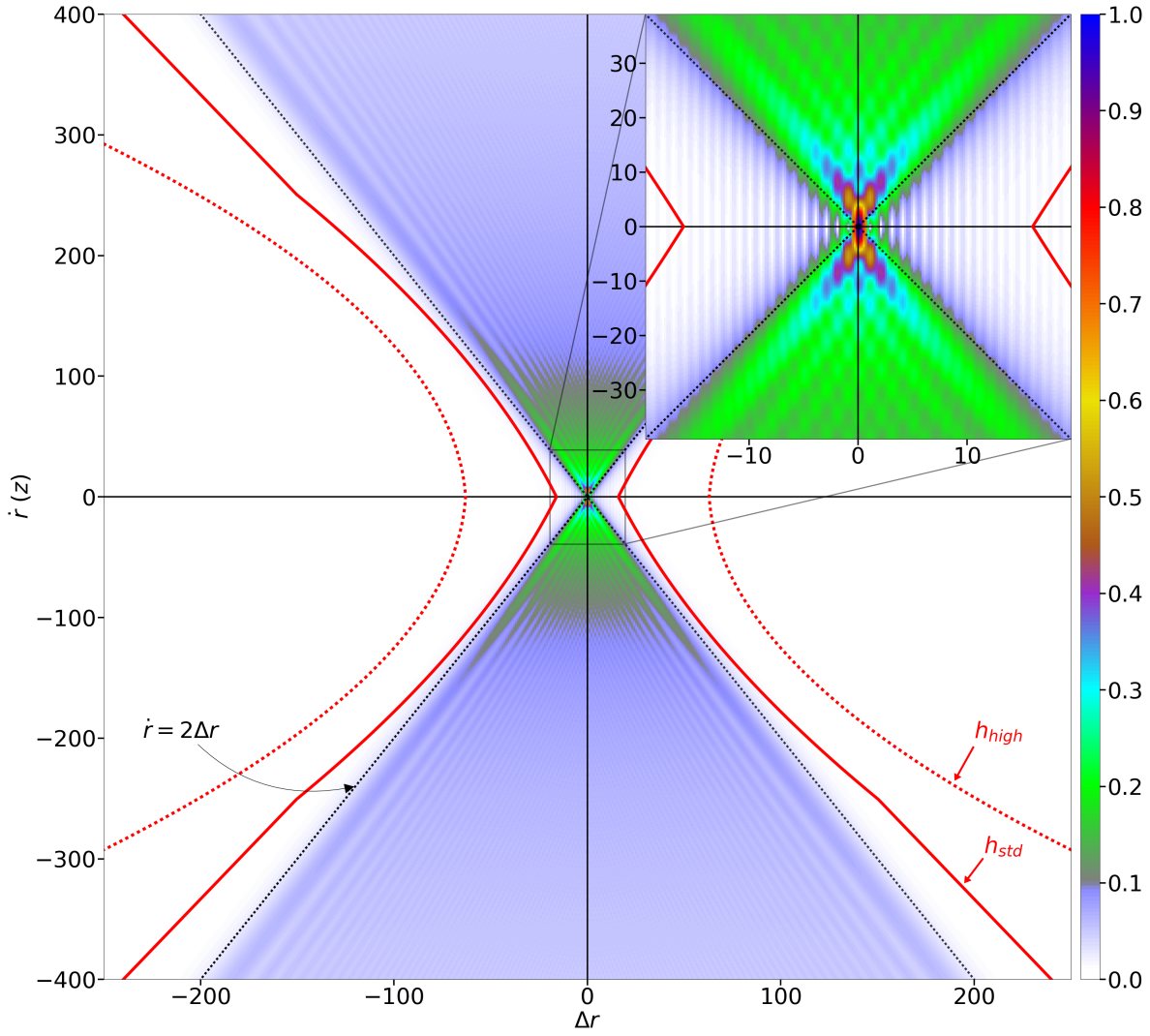


Figure 2.7: The magnitude of the complex acceleration filter coefficients. These are the terms that $\mathcal{F}_k$ are multiplied with, in Equation 2.9. The magnitude of these values drops off to close to zero, for $|\Delta r| \gtrsim 0.5|\dot{r}|$. The two sets of red lines mark the boundary widths of the two acceleration filters used in `Accelsearch`. The actual filter widths are the horizontal distance between the two relevant lines, which are a function of $\dot{r}$ (Equation 3.2). The solid red lines mark the standard accuracy, and the dashed lines mark the high-accuracy filter lengths.

$|\dot{r}| = 2\Delta r$ quickly drop off to close to zero. Thus, the response will be close to fully corrected if m is chosen, such that $|\dot{r}| < m \lesssim |2\dot{r}|$ [115].

Figure 2.8 shows how frequency domain acceleration correction can effectively reverse the smearing caused by a change in frequency. The black line in the top panel is a section of the normalised, fine-binned power spectrum of some Gaussian noise. The dashed magenta line is a similar power spectrum of the same noise, with the addition of a weak periodic signal which has a constant change in frequency. The mean frequency of the signal is 95.45 Hz (the dashed green line) and the change in frequency is $0.14\,\mathrm{Hz}$. In the power spectrum, this is a signal centred on bin 51247.55 which drifts by 75 bins. It can be seen that around the central bin, the magenta line has some additional power. Unlike the strong signal in Figure 2.6, none of the peak powers in this region are significantly higher than those of background noise. Thus, this weak signal would not be detected by a standard periodicity search. The green line in the bottom panel represents the same data as the magenta power spectrum, corrected for the appropriate acceleration using the correlation method with $m = 120$. The clear spike in this power spectrum would most certainly result in an 11σ detection of the periodic signal. This spike is at the appropriate frequency and power, as is evident by its similarity to the blue dashed line. This line is the spectral peak of the same noise with the addition of a similar periodic signal with no change in frequency. This is a clear demonstration of the ability of the correlation method to reverse the smearing caused by a change in frequency.

A feature that plays a key role in the FDAS is that of r-$\dot{r}$ planes. The standard approach in the FDAS is to calculate a collection of corrected Fourier components across an evenly spaced grid of r and $\dot{r}$ values. These Fourier components are referred to as an r-$\dot{r}$ plane, and the powers of these values form a 2D power spectrum. A high resolution example of such a 2D power spectrum, containing the signal from a binary pulsar, is shown in Figure 2.9. These types of planes play a large role in this work and, as a matter of convention, r is always considered the horizontal axis and $\dot{r}$ as the vertical. Harmonic summing can be performed with these 2D power spectra, in which case the value of both r and $\dot{r}$ are scaled by the harmonic number. These harmonically summed 2D spectra can be searched for significant values in a similar manner to the standard frequency domain search for isolated pulsars.

The efficiency of the FDAS is a result of all the information of a periodic signal being localised

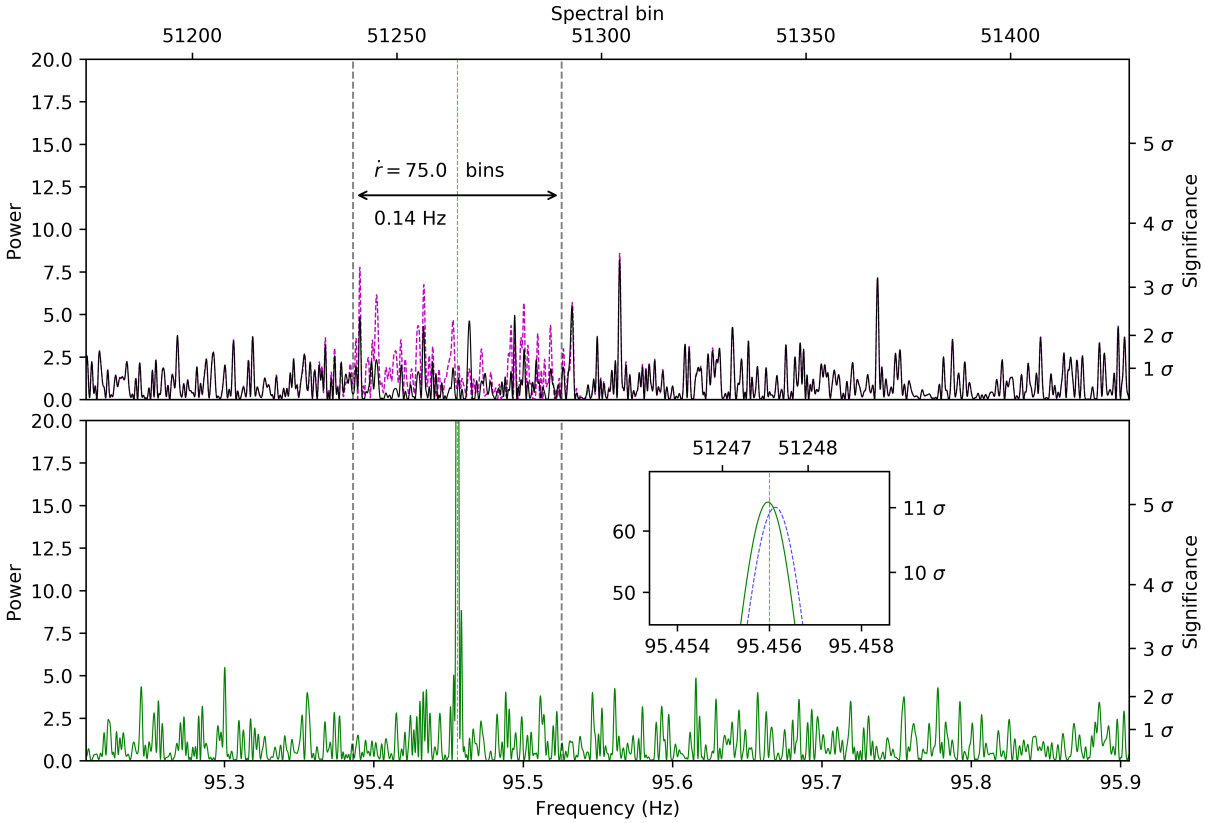


Figure 2.8: An acceleration corrected power spectrum. The black line in the top panel is the normalised power spectrum of Gaussian noise. The dashed magenta line is the power spectrum of the same noise, with an additional weak periodic signal. This signal has mean frequency of 95.45 Hz ($r = 51247.55$) and a constant change in frequency that shifts by $0.14\,\mathrm{Hz}$ ($\dot{r} = 75$). This changing frequency smears the power of the signal over a number of spectral bins; this additional power is localised to the region surrounding the mean frequency. The green line in the bottom panel is the acceleration-corrected power spectrum of the same data as the magenta line. This spectrum shows a clear peak at the central frequency. The blue dashed line shown in the inset is the power spectrum of the noise and a similar periodic signal with no change in frequency. The similarity of the two shows that the acceleration correction successfully recovers the power lost due to the constant change in frequency.

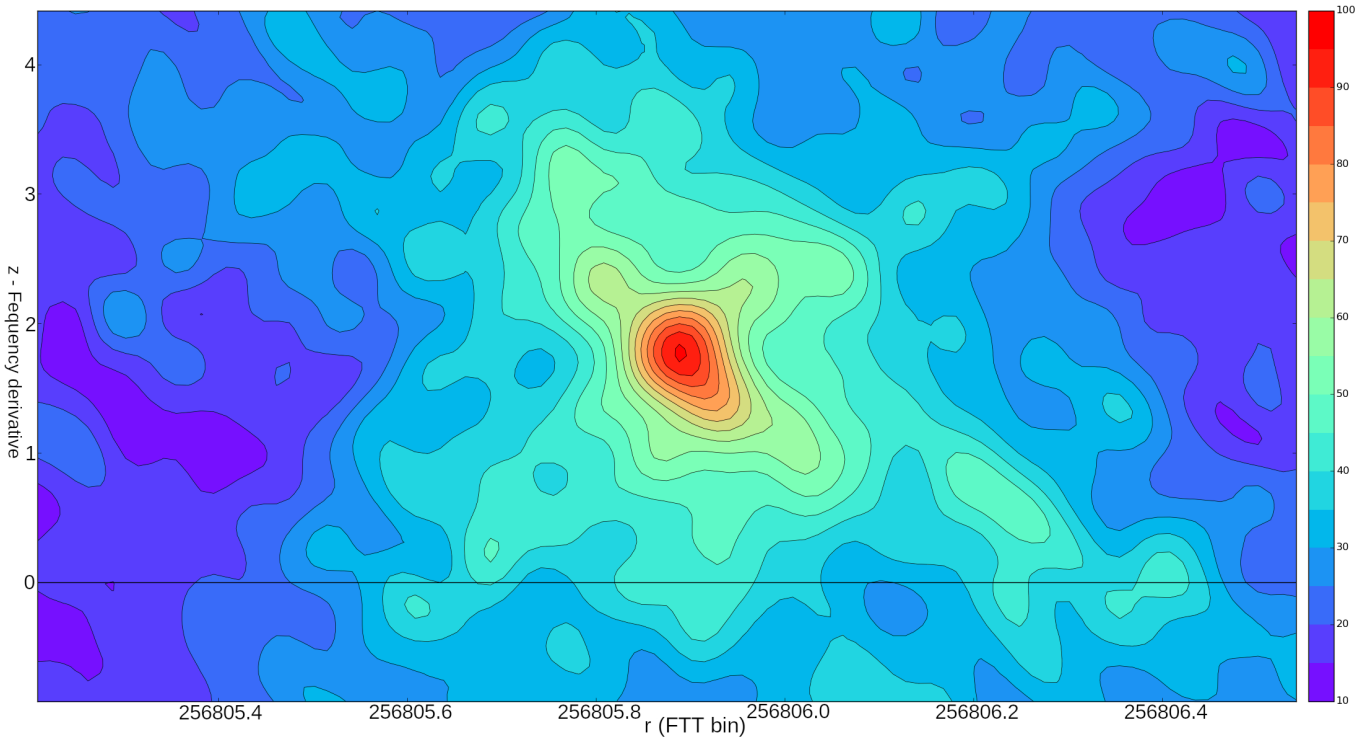


Figure 2.9: This r-$\dot{r}$ plane is a section of a 2D power spectrum containing the signal from the fundamental harmonic from a 4 hour observation of a pulsar in a binary orbit.

in the frequency domain. It is clear that in Figure 2.8 most of the information associated with the periodic signal is centred on the mean frequency and that the range of this information is limited to an area of similar width to the change in frequency. Thus, the acceleration correction can be performed with a relatively short filter and a small number of memory-local values. In the FDAS, the acceleration parameter is iterated over after the full time series has been Fourier transformed, meaning that only one long FT is required per DM trial. By contrast, in the time domain, the information of the signal is spread across the entire time series. Thus, the TDAS requires resampling the entire time series and performing a full FT for every acceleration trial. This makes performing a FDAS more computationally efficient than a TDAS.

The implementation of this method is not only more efficient than the time domain equivalent, it has a number of levels of data and task parallelism and is well suited to GPU acceleration. At the time of writing, the FDAS had not been implemented on the GPU, thus a GPU-accelerated version of this FDAS was chosen as the focus of this work.

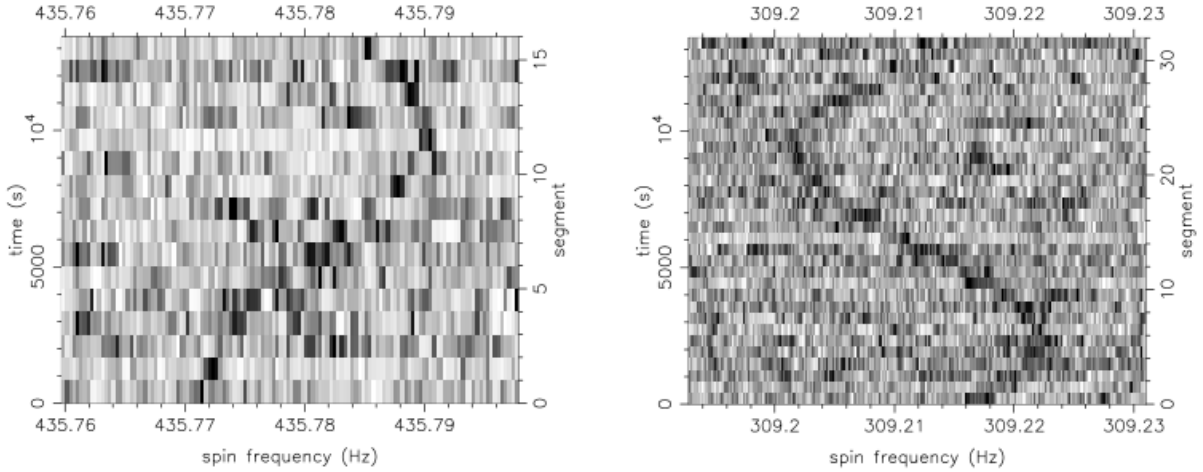


Figure 2.10: Dynamic power spectrum from two binary pulsars. Here, the change in frequency can be seen as a sinusoidal shift in frequency, as a function of time. (Image taken from [22])

2.3.4.3 Sideband search

If $T_{\rm obs} \gtrsim 1.5 \times P_{\rm orb}$, another technique – the *sideband search* – can be used [52, 114]. This method relies on the fact that the power spectrum of a binary pulsar has a characteristic shape, as depicted in Figure 2.6. This is a result of the modulation of the observed spin frequency with the $P_{\rm orb}$. This causes sidebands to appear around $f_{\rm spin}$; these sidebands can be detected with a short FT. Thus a number of short FFTs can be used to detect $P_{\rm orb}$ [114]. This search method relies on the period of the orbit being significantly longer than the observation. With the integration time of most blind searches being in the realm of minutes, this would require an extremely tight orbit, well beyond what is generally seen. Thus, this method is rarely, if ever, used in large-scale surveys, however it has some merit in longer-duration targeted observations.

2.3.4.4 Dynamic power spectrum searches

Acceleration searches are effective when $T_{\rm obs} \lesssim 0.1 \times P_{\rm orb}$, while the sideband search is effective when $T_{\rm obs} \gtrsim 1.5 \times P_{\rm orb}$. This leaves a "sensitivity gap" when $T_{\rm obs}$ is comparable to $P_{\rm orb}$. In this range, some form of DPS search can be used. With this technique, a dedispersed time series is split into many small sections, each of which is separately Fourier-transformed and potentially harmonically summed. A collection of such power spectra can be thought of as 2D, frequency-versus-time data, and is referred to as a dynamic power spectrum (DPS).

The orbitally modulated signal from a pulsar will appear as a curved feature or wavy track in a DPS. When an orbit has a large eccentricity, the tracks can be fairly complex. Six parameters are needed to describe these complex tracks: $f_{\rm spin}$ and the five orbital parameters. Thus, a pulsar search aiming to detect these tracks in a DPS covers the same nine-dimensional parameter space as the fully coherent search discussed at the beginning of this section. However, the use of power spectra makes this an incoherent method, meaning it will be less sensitive than the fully coherent time domain resampling method. When an orbit has a low eccentricity, as is seen in most binary MSPs, the track will be sinusoidal. The sinusoidal tracks of two such pulsars are shown in Figure 2.10. These sinusoidal tracks can be described using four parameters: $f_{\rm spin}$, amplitude, phase, and wavelength[7]. Most DPS methods focus on detecting sinusoids.

Methods based on an incoherent DPS may be less sensitive than the fully coherent search. However, in the applicable range, they are still more sensitive than the acceleration and sideband searches. The DPS methods have the advantage that incoherent addition allows the use of multiple observations, whereas the acceleration searches and the sideband search rely on coherent data and are thus limited by the maximum observation length. The use of multiple observations can increase the sensitivity, at the cost of additional computation.

DPS searches have been used very rarely. Detecting tracks in a DPS is not a trivial task, as they are usually very weak, with the individual points of a track only slightly above the level of background noise. Lyne et al. [71] performed a 1.6 hour observation of the globular cluster Terzan 5, and as part of a search, a number of dynamic power spectra were visually inspected for the tracks of pulsars [71]. This type of visual inspection is only feasible for targeted searches and is impractical for large-scale blind searches. There are two notable cases in the literature in which an automated DPS search has been used. Chandler [22] developed a hierarchical scheme to search dynamic power spectra and employed it to discover three faint pulsars in M62. Aulbert [5] developed a novel method that employs a hierarchical 2D Hough transform to detect sinusoids in a DPS [5, 6].

A Hough transform is a procedure that can be used to detect a parametric feature in an image, it is traditionally used to detect lines or circles. It works by transforming points from the image into corresponding multi-dimensional surfaces in a parameter space. In an implementation of a Hough

[7]The wavelength of the sinusoidal track is $P_{\rm orb}$ and it will oscillates about $f_{\rm spin}$.

transform, the surfaces representing the significant points in the image are summed together in a discrete parameter space. A significant feature in the image space will result in a locus in the parameter space. Thus, the detection of a locus in the parameter space allows the detection of a feature in the image.

In the case of the pulsar search employed by Aulbert [5], the input image is a DPS and the parameters describe a sinusoid. Due to computational constraints, the Hough transform developed here was a hierarchical 2D transform rather than a full 4D transform. As part of our early work, a GPU-accelerated implementation of the full 4D sinusoidal Hough transform was developed [60]. This implementation was 20 to 25 times faster than our CPU equivalent, however, even with GPU acceleration this method is still too time-consuming to be used in wide-scale searching.

2.3.5 Candidate selection

The final component of the pulsar search pipeline is candidate selection. This step takes the pulsar candidates produced by the periodicity search and collates the relevant details of each candidate. This information is usually presented as a plot and includes an integrated pulse profile, obtained by folding the data. Calculating pulse profiles for many candidates can be computationally intensive [19, 66] and folding is becoming one of the most computationally demanding tasks in modern pulsar searches [66]. Traditionally, the candidate plots are inspected by a human operator, usually a graduate student, to identify viable candidates. More recently, machine learning systems have been used for this purpose [34, 65]. Viable candidates are usually scheduled for reobservation, confirmation, and precision timing.

Chapter 3

CPU implementation of the FDAS

This chapter describes the existing serial CPU implementation of the FDAS. This implementation is the binary `Accelsearch`, which is found in version 2.0 of the pulsar search suite PRESTO [112]. `AccelGPU`, our GPU-accelerated FDAS is a port of `Accelsearch`, and is designed to fit into the PRESTO ecosystem. The methodological analysis of the existing implementation in this chapter should give the reader an understanding of the key parameters, basic functioning, and components of the FDAS. An understanding of these is necessary for the discussion about the parallel decomposition and GPU acceleration of the FDAS which is presented in Chapter 5.

The FDAS is a component of the greater pulsar search pipeline (Section 2.3). It takes as input the DFT of a dedispersed time series and aims to detect periodic signals that exhibit a close-to-linear change in frequency. It uses the correlation method (Section 2.3.4.2) to correct for a constant change in frequency and coherently "sweep" all the power into a single spectral bin.

Before going into the technical details of the implementation, we give a brief outline of the range and resolution of the search parameters explored by `Accelsearch`: f_{spin}, a, and h. As `Accelsearch` operates in the frequency domain, it expresses frequency and acceleration in terms of spectral bins. Frequency is expressed in spectral bin index (r) with the frequency of the r^{th} bin given as: $f_r = r/T_{\text{obs}}$. Acceleration is expressed as the number of spectral bins by which the frequency of a periodic signal changes during an observation ($\dot{r} = \dot{f}\ {T_{\text{obs}}}^2$), which is usually represented with the symbol Z. In `Accelsearch`, the command line parameter Z_{max} is used to control the number of acceleration trials (N_Z) investigated. The application examines acceleration trials from -Z_{max} to +Z_{max} at a spacing of two bins, and centred on zero. Thus, the number of acceleration trials is given as: $N_Z = 1 + 2\lceil Z_{\text{max}}/2 \rceil$. The default frequency range explored is from $1\,\text{Hz}$ to $10\,000\,\text{Hz}$ at a frequency resolution of 0.5 bins. In `Accelsearch`, the maximum number of harmonics searched is specified as one, two, four, eight or sixteen. These correspond to the stages of harmonic summing (Section 3.4.3) and a search will explore each stage up to the

maximum specified. In this way, the application segments the search parameters covered by the FDAS.

3.1 Accelsearch pipeline

The components and structure of `Accelsearch` are shown in Figure 3.1. This implementation breaks the FDAS into two stages: *candidate generation (CG)* and *candidate optimisation (CO)*. The first, CG, performs a search of a large fixed resolution r-$\dot{r}$ space containing in the order of 10^{7-12} points. It produces a collection of potential candidates, referred to as *initial candidates*; these are distinct from *final candidates*, which are the output of the CO stage. The CO stage refines the r and $\dot{r}$ values of the candidates found in the CG stage.

The CG stage creates sections of r-$\dot{r}$ plane, harmonically sums these, and identifies potential candidates in the summed spectra. There are two variants of the CG stage: *standard* and *in-memory*. These variants differ in the type of r-$\dot{r}$ plane employed for harmonic summing. Both variants are iterative in nature, and in both cases, each iteration consists of the same two basic tasks. The first is *plane creation* and the second comprises harmonic summing and searching. The number of times the plane creation task is performed differs between the standard and in-memory variants; this task is discussed in more detail in Section 3.2.

The standard variant is used when the size of the r-$\dot{r}$ space to be searched exceeds the available memory. This variant separates the input DFT into a number of small sections, referred to in this work as *segments*. The bulk of the CG stage consists of a loop that iterates over these segments. Each iteration creates a separate section of r-$\dot{r}$ plane for each harmonic summed, meaning that up to sixteen sections of r-$\dot{r}$ plane are created for each segment. Creating each of these small sections of r-$\dot{r}$ plane requires one instance of the plane creation task to be performed. Harmonic summing is performed in stages (Section 3.4.3), using these small sections of r-$\dot{r}$ plane. After each stage of harmonic summing, the summed powers are searched for significant values which, if found, are stored as initial candidates. Thus, all of the tasks previously mentioned are performed in each iteration, with the plane creation task being repeated for each harmonic summed. The benefit of the standard variant is that each segment can be processed independently, using only a small amount of working memory to hold the r-$\dot{r}$ planes. However, the disadvantage of this variant is that each of the planes is a sub-section of one or more of the planes of some other segment;

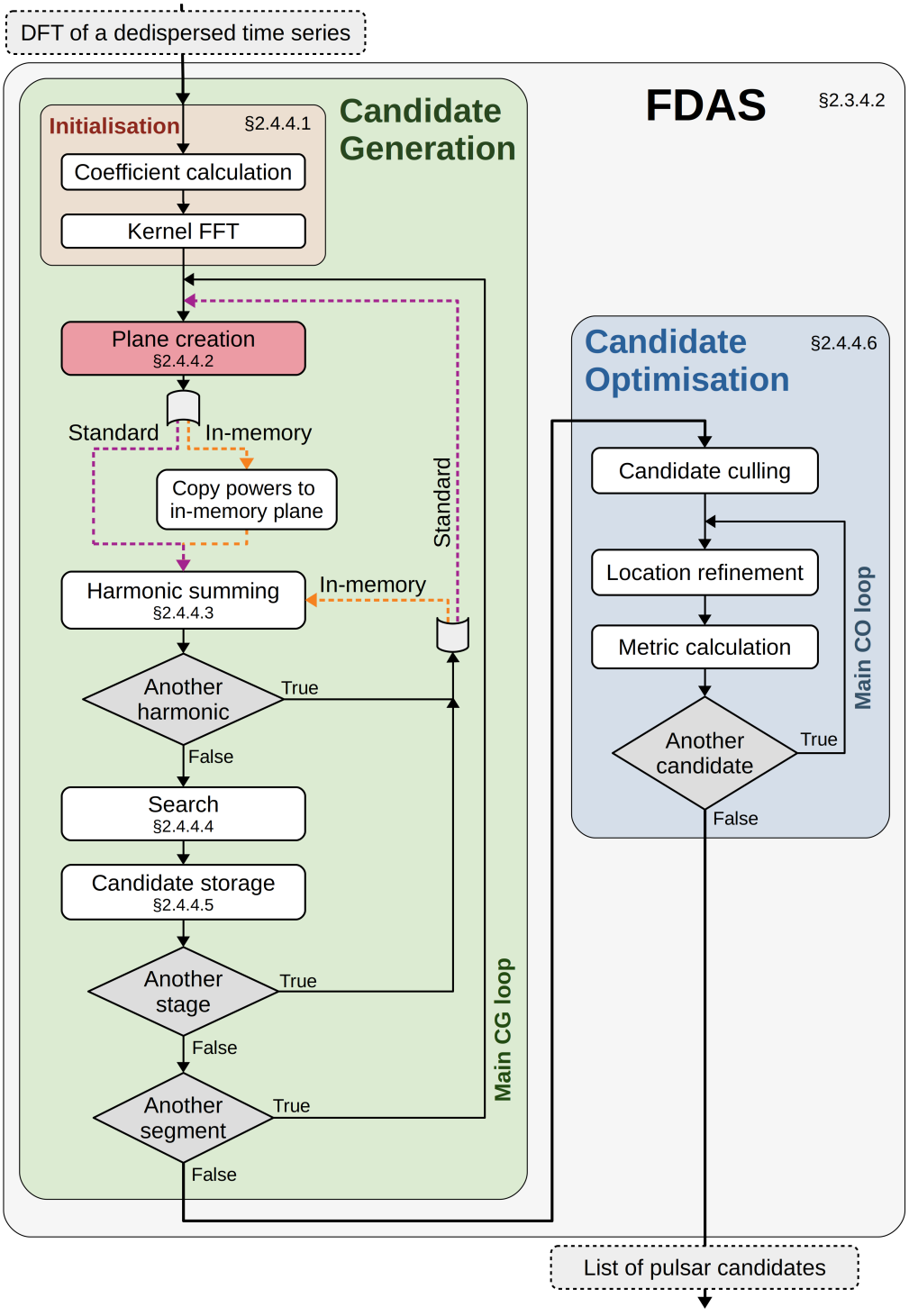


Figure 3.1: A schematic representing the `Accelsearch` pipeline, which is an implementation of the FDAS. This search takes the DFT of a dedispersed time series as input and produces a list of pulsar candidates. The search is broken into two stages, both of which are iterative in nature, with each iteration consisting of a number of small components. The candidate generation stage has two variants, standard and in-memory, which differ in the number r-$\dot{r}$ of planes created in each of the iterations of the main candidate generation loop.

therefore, this method duplicates a moderate amount of computation. This can be avoided if there is sufficient memory to store all the r-$\dot{r}$ values required for this stage of the search. Thus, this variant is only used when available memory is limited.

The in-memory variant of the CG stage is used when there is sufficient memory to store all the r-$\dot{r}$ powers used in the CG stage. In this variant the *full* r-$\dot{r}$ *plane* is built up incrementally using the overlap and save technique [17, 35, 109]. This entails iterating over the input DFT in segments, similar to the standard variant. However, the in-memory variant creates only a single section of r-$\dot{r}$ plane for each segment. In each iteration the relevant powers from this small section of r-$\dot{r}$ plane is copied to the appropriate location in a large memory space, which is set aside to hold the full r-$\dot{r}$ plane. This full plane is built up incrementally from low to high frequency; allowing each iteration to perform harmonic summing and searching using the powers stored in the partially filled plane. This, however, means that each iteration is dependent on the completion of all previous iterations. As with the standard variant, each iteration performs all the basic tasks; although the in-memory variant creates only a single section of r-$\dot{r}$ plane per iteration. This variant leverages memory to store these r-$\dot{r}$ values and thus reduces duplicated computation. As a result, this variant is faster than the standard variant when summing more than one harmonic.

Creating a small section of r-$\dot{r}$ plane is one of the key tasks in both variants of the CG stage. It is computationally intensive and can be broken down into a number of components. This important task thus warrants further attention.

3.2 Creating a section of r-$\dot{r}$ plane

Calculating a small discrete portion of r-$\dot{r}$ plane is one of the fundamental tasks in the FDAS. Section 2.3.4.2 describes how a single r-$\dot{r}$ point can be calculated using Equation 2.9. This calculation can be thought of as the convolution of some DFT components, with an acceleration filter. This filter is acceleration-specific and is thus the same for all points in a single row of an r-$\dot{r}$ plane. This means that an entire row can be calculated with the convolution of some DFT components with an appropriate acceleration filter. Long convolutions such as these are usually calculated using the computationally efficient *fast convolution* technique.

3.2.1 Fast convolution

Fast convolution is an efficient means of calculating large convolutions using FFTs. The convolution theorem states: the Fourier transform of the convolution of two functions is equivalent to the product of their individual Fourier transforms. Thus, the convolution of two functions can be calculated as the inverse Fourier transform (iFT) of the product of the FTs of the two functions:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

In the case of a discrete convolution, f and g are discrete vectors of length ω. If ω is a power of two, the ω-point convolution can be calculated using the efficient FFT and inverse fast Fourier transform (iFFT). This method of calculating convolutions using FFTs and iFFTs is referred to as *fast convolution*. This allows a discrete ω-point convolution to be calculated with three transforms of complexity $\mathcal{O}(w \log w)$ and a pointwise multiplication ($\mathcal{O}(w)$).

The FT of the filter is referred to as a *convolution kernel*, as it can be calculated once and used to perform many convolutions. An understanding of the layout of the filter coefficients used to generate a convolution kernel is important as it affects the usability of values generated using the convolution kernel. A convolution kernel is created by arranging filter coefficients in a wrapped fashion, meaning that the centre of the filter ($\Delta r = 0$) is located at the beginning of the vector. The first half of the vector is filled by the right-hand wing of the filter, while the left-hand wing is shifted so that it fills the last half of the vector. This wrapped layout is shown in the bottom panel of Figure 3.2. A vector containing wrapped filter coefficients is Fourier-transformed to produce a convolution kernel.

The convolution theorem makes two assumptions that do not hold true in the case of generating r-$\dot{r}$ values. The first is that the length of the signal and the filter are the same; the second is that the signal and the filter are both periodic. In `Accelsearch`, the length of the signal is the width of the r-$\dot{r}$ plane (ω), while the length of the filter (m) is proportional to $\dot{r}$. The maximum acceleration trial is determined by the $Z_{\max}$ parameter, which is generally an order of magnitude smaller than ω; thus the filter is significantly shorter than the signal. This disparity in length is overcome by zero-padding the vector containing the acceleration coefficients. This padding is added to the centre of the vector, so that the non-zero left and right wings of the filter are on either

end of the vector – as is depicted in the bottom panel of Figure 3.2.

The second assumption – that the signal and filter are periodic – is what allows the filter coefficients to be wrapped. However, the signal not being periodic means that a number of the results will be "contaminated" as the convolution will contain some values from the opposite end of the signal. This type of wraparound contamination will be present in as many results as there are non-negative values in the left- and right-hand wings of the filter. If there are $m/2$ non-zero filter coefficients on either end of the wrapped vector, the first and last $m/2$ values of the output will be contaminated by wraparound effects, as depicted in Figure 3.2.

The fast convolution described above can be used to efficiently calculate the acceleration-corrected Fourier components for a single acceleration trial. These make up a single row of an r-$\dot{r}$ plane.

3.2.2 2D plane creation

To calculate a discrete 2D grid of r-$\dot{r}$ values, a number of convolution kernels of similar length are generated. Each of these kernels is pointwise multiplied with the FT of an equivalent length section of the input DFT. The inverse discrete Fourier transforms (iDFTs) of these products make up the rows of a 2D grid of r-$\dot{r}$ values. This "raw" r-$\dot{r}$ plane contains some results that are contaminated by the wraparound effect; Figure 3.3 depicts the layout of a raw r-$\dot{r}$ plane. The width of a raw plane (ω) is chosen to be a power of two to allow for the use of FFTs, while the height is the number of acceleration trials, which is determined by $Z_{\max}$. The amount of contamination at the edges of the plane is proportional to $\dot{r}$, and is shown by the blue sections in Figure 3.3. The width of the widest rectangular region that does not contain contamination (u) – the dark green section in the figure – is determined by the widest filter used in the correlation kernels. The term *half-width* (t) refers to half the width of the widest filter used in the correlation kernels. A rectangular plane of uncontaminated r-$\dot{r}$ values, of arbitrary width S_w, can be created by scaling ω, although ω is increased in powers of two. Thus, the actual uncontaminated values calculated may contain some padding (p). Formally, the relationship between these is given as:

$$\omega = S_w + 2t + p \tag{3.1}$$

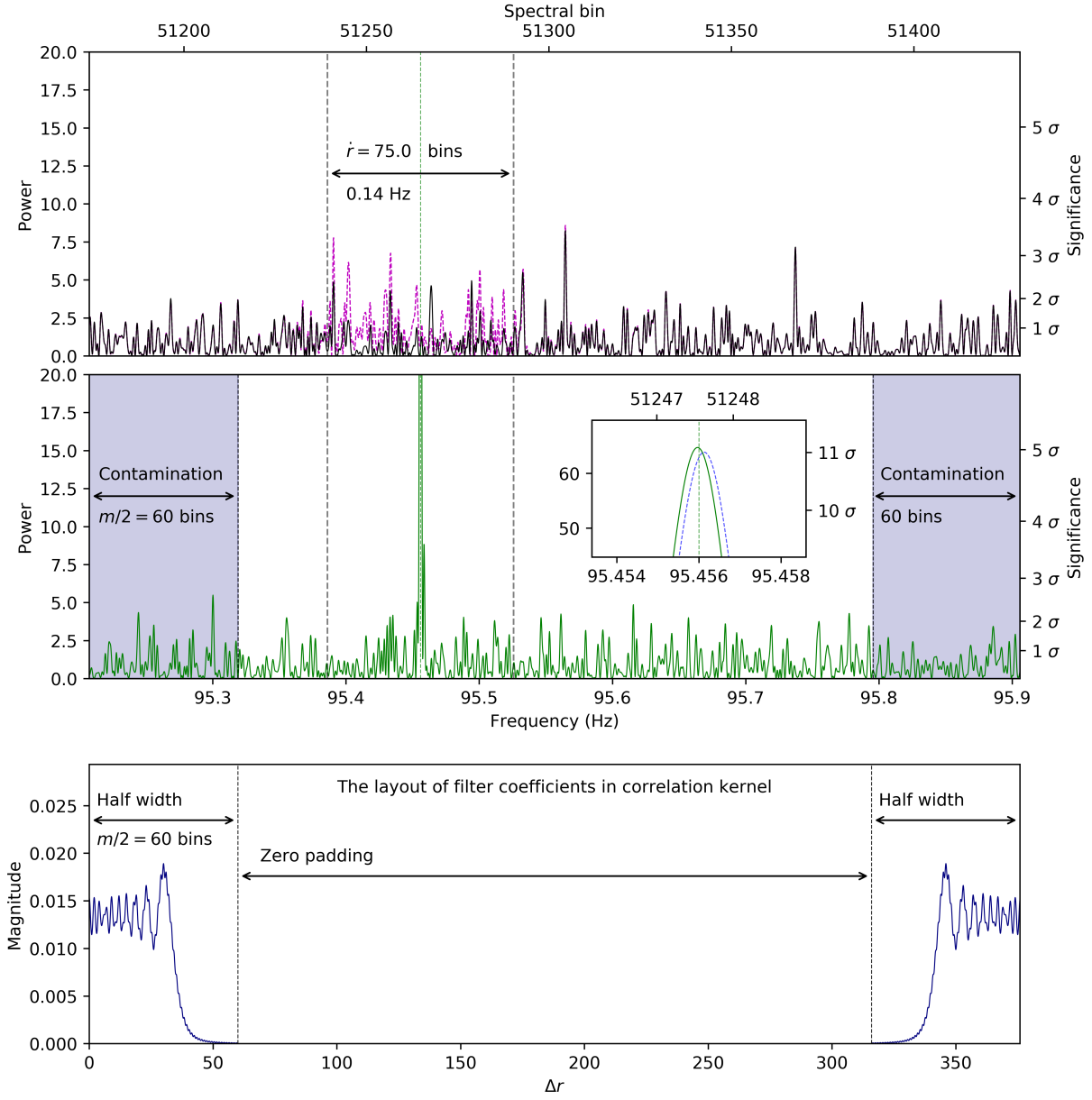


Figure 3.2: The bottom panel shows how filter coefficients are arranged in a wrapped fashion, with the right-hand wing of the filter on the left and the left-hand wing on the right. The region between these two wings is padded with zeroes to make the correlation kernel the same length as the signal. The values that are to be multiplied with the correlation kernel are not periodic. This means that the ends of the output that correspond to the non-zero filter coefficients will be contaminated as a result of the wraparound effect. The contaminated regions are shown as blue shaded areas in the middle panel and the values in these regions will be incorrect and cannot be used.

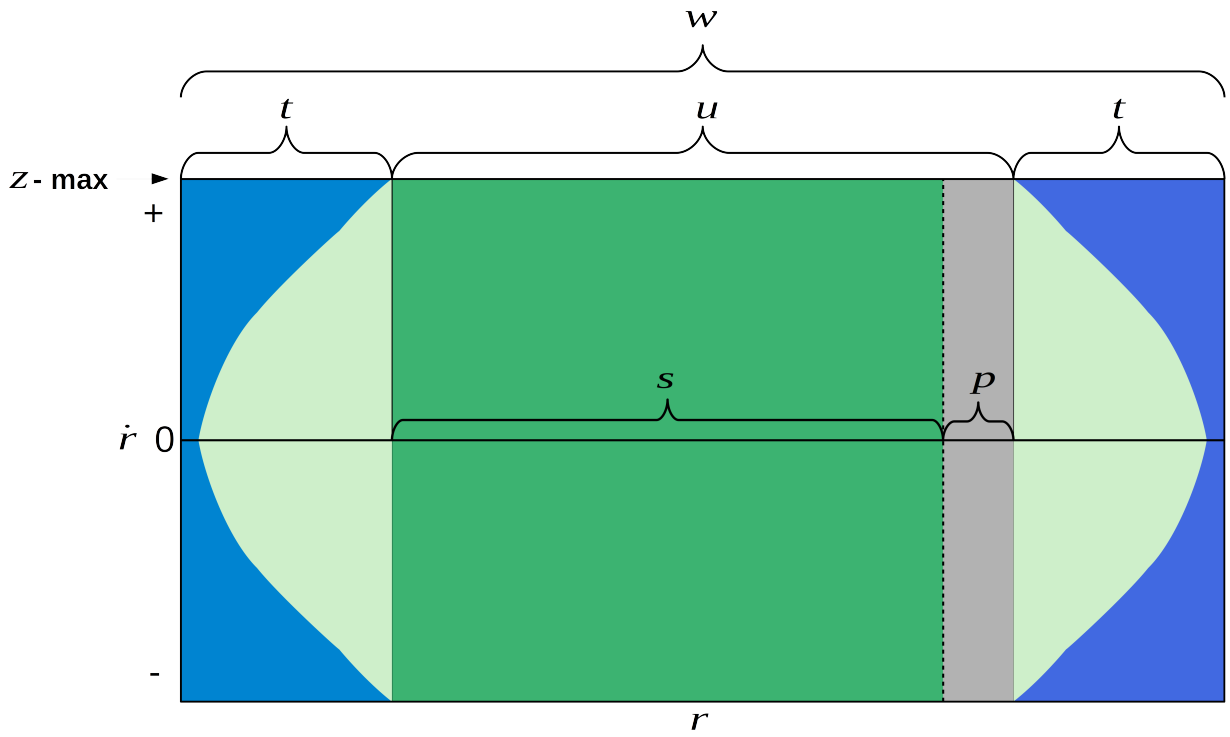


Figure 3.3: The layout of a 2D convolution kernel, which is used to generate a section of raw r-$\dot{r}$ plane. The blue areas mark the locations that are contaminated by wraparound values, while the usable values are green. The dark green section has width u, and is the widest usable rectangular section. t is the half-width of the plane, which represents half of the width of the widest acceleration filter used to generate the kernel, and is found at the maximum acceleration trial ($Z_{\max}$). ω is the plane width and is a power of two, while S_w is the segment size: the width of each iteration over the input DFT. If $S_w < u$, some padding (p) may be needed to ensure that ω is a power of two.

3.3 Configuration parameters

We separate the parameters of a search into two types. Search parameters relate to the properties of a pulsar, such as f_{spin} and h, while *configuration parameters* determine how a search is performed. This section describes a few of the key configuration parameters, specifically those that have a substantial impact on performance.

3.3.1 Plane size and resolution

The size and resolution of the r-$\dot{r}$ planes created in the CG stage warrant some attention. All the planes created in this stage have the same r and $\dot{r}$ resolution. The height of the primary planes

is N_Z, and all planes have an $\dot{r}$ resolution of 2 bins. The width and frequency resolution are slightly more complicated. Using the fast convolution technique it is possible to create a plane with a frequency resolution of $1/r_n$ where $r_n \in \mathbb{N}^+$. In `Accelsearch`, the frequency resolution used is 0.5 bins[1], i.e. $r_n = 2$. This fine binning is achieved by spreading and zero-padding the input values and creating the filter coefficients at the desired resolution. Spreading and zero-padding is done so that each input value is separated by $r_n - 1$ zeroes. It is noteworthy that ω represents width, measured in number of values not number of bins. Thus, the number of input DFT components needed to generate a raw r-$\dot{r}$ plane of width ω is ω/r_n. Similarly, the number of input DFT components in each segment of the main loop of the CG stage is actually S_w/r_n. In `Accelsearch`, the values of both r_n and S_w are hardcoded. The number of r-$\dot{r}$ points created is directly proportional to r_n, thus the run time of the CG stage is directly proportional to r_n. Having S_w hardcoded can have a significant impact on performance and is discussed in detail in Section 5.2.

3.3.2 Accuracy

The accuracy of an acceleration correction is in part determined by the width of the filter used. The effects of the acceleration are spread across all bins of the spectrum, meaning that the true width of the acceleration filter is infinite. However, the magnitude of the filter coefficients drop to close to zero for $|\Delta r| \gtrsim |\dot{r}|/2$ (Figure 2.7). This shows that the majority of the spectral power of the accelerated signal is localised in a small number of bins surrounding the central frequency of the signal. Accordingly, the filter can be clipped at some width m and considered finite. Thus, the filter width has some influence on the accuracy of the acceleration correction, with longer filters being more accurate. Conversely, for a set accuracy, $m \propto \dot{r}$. `Accelsearch` uses two set accuracies, referred to as *standard* and *high*. Standard-accuracy filters are used to generate the sections of r-$\dot{r}$ plane used in the CG stage, while high-accuracy filters are predominantly used in the final steps of candidate optimisation. The filter lengths used in `Accelsearch` are given in Equation 3.2. For large $\dot{r}$ values, m is linearly proportional to $\dot{r}$, with high-accuracy filters being twice the width of standard-accuracy filters. For smaller $\dot{r}$ values, the relationship is quadratic with a non-zero minimum. This means that for low acceleration, the filter widths are wider than

[1]The points at these half-bin locations are calculated using filters of appropriate length (Section 3.3.2). This is opposed to the alternative of interbinning the acceleration-corrected Fourier components at integer Fourier frequencies.

the base linear relationship. The accuracy is consequentially increased for low accelerations, where the majority of pulsars are found. The two filter widths are shown by the solid and dashed blue lines in Figure 2.7. The width of the filter used at a specific $\dot{r}$ is the horizontal distance between the relevant lines. In Figure 2.7, it can clearly be seen that both accuracies encompass all the filter coefficients with significant magnitudes. The widening of the filters for small $\dot{r}$ values is also clearly visible. Equation 3.2 also determines the boundary of the contaminated blue sections in Figure 3.3.

$$
\begin{aligned}
m_{\mathrm{std}}(\dot{r}) &= 2 \times \begin{cases} 0.00089\,\dot{r}^2 + 0.3131\,\dot{r} + 16 & \text{if } \dot{r} \leq 250.63 \\ 0.6\,\dot{r}, & \text{otherwise} \end{cases} \\
m_{\mathrm{high}}(\dot{r}) &= 2 \times \begin{cases} 0.002057\,\dot{r}^2 + 0.0377\,\dot{r} + 63 & \text{if } \dot{r} \leq 504.316 \\ 1.2\,\dot{r}, & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.2}
$$

3.4 FDAS components

Here, we outline the main components of the `Accelsearch` processing pipeline. These are shown in Figure 3.1 and 3.4, and are discussed in some detail in the following sections.

3.4.1 Initialisation

The CG stage has a once-off initialisation during which the convolution kernels are generated. This is done by Fourier transforming an array containing acceleration filter coefficients. Calculating these coefficients is computationally intensive and requires many trigonometric calculations and the evaluation of two pairs of Fresnel integrals; this calculation is discussed in detail in Section 6.1.1. The convolution kernels can, however, be reused and thus need only be computed once per search. A section of r-$\dot{r}$ plane is generally considered a unit, to be created and processed as one entity. Thus, this work will often refer to the collection of all of the individual convolution kernels used to create a section of r-$\dot{r}$ plane as "the" convolution kernel. The main steps involved in creating a convolution kernel for an r-$\dot{r}$ plane section are given in Algorithm 1. The computationally expensive components are calculating the filter coefficients and performing the

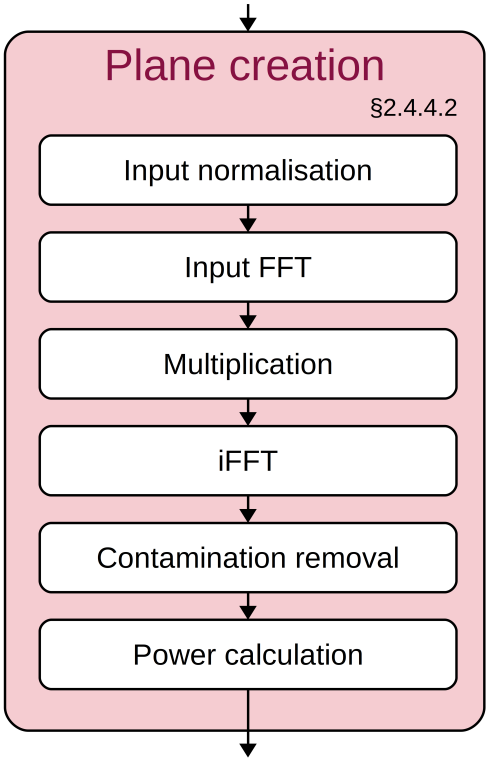


Figure 3.4: The components of the plane creation task.

FFTs.

Algorithm 1 CPU kernel creation

1: Choose ω the width of the f-$\dot{f}$ plane
2: **for** each value of $\dot{r}$ **do**
3: Determine filter length (Equation 3.2)
4: Calculate the coefficients of the acceleration filter of length m
5: Centre, wrap and zero-pad filter coefficients in an array of length ω
6: FFT the resulting array
7: Store the complex result

3.4.2 Plane creation

The plane creation task is repeated hundreds to thousands of times in a search. This task consists of six fundamental components: input normalisation, input FFT, multiplication, iFFT, contamination removal, and power calculation. These are shown in Figure 3.4 and the procedure is detailed in Algorithm 2.

Some form of normalisation is necessary to be able to accurately compare, sum, and search separate sections of r-$\dot{r}$ plane. In `Accelsearch`, this is achieved by normalising each section

Algorithm 2 CPU plane creation

1: Extract the relevant ω/r_n values from the input DFT
2: Calculate the median of the powers of these values
3: Normalise using the median
4: FFT these values
5: **for** each trial value of $\dot{r}$ **do**
6: Pointwise multiply the prepared input with the relevant row of the kernel
7: iFFT the resulting array
8: Extract the uncontaminated r-$\dot{r}$ values
9: Calculate and store the powers in memory

of the input DFT prior to using them to generate a portion of r-$\dot{r}$ plane. The objective of this normalisation is for the mean power to be one. Rather than using standard mean normalisation, median normalisation is used as it is insensitive to high power outliers, including those often introduced by radio frequency interference (RFI). Median normalisation entails dividing each complex value by the product of $ln(2)$ and the median of the powers of the DFT section [115]. Once the section of input data has been normalised, it is FFTed, to produce a *prepared* section of input. The rows of a convolution kernel are all the same width, thus a single section of prepared input can be used to generate a single section of r-$\dot{r}$ plane; this means that only one instance each of the input normalisation and FFT components are required per plane.

The prepared input is then pointwise multiplied with each row of a precalculated convolution kernel. Each one of these rows is then iFFTed to produce a raw r-$\dot{r}$ plane. The relevant uncontaminated values are extracted from each row. These sections are all the same width, so as to produce a rectangular section of r-$\dot{r}$ plane. Finally, the power of each of the extracted values is calculated. This entire procedure is performed using a small section of working memory. In the in-memory variant, these powers are copied from working memory to the relevant location in the full r-$\dot{r}$ plane.

3.4.3 Harmonic summing

Incoherent harmonic summing can be used to increase the sensitivity of a search to pulsars with low duty cycles. The standard method of implementing harmonic summing uses stages, and "stretches" smaller planes to add them onto larger ones. This method may seem slightly counter-intuitive at first, but it has a number of advantages over the naïeve implementation. There are usu-

ally up to six stages in which one, two, four, eight, sixteen or thirty-two harmonics are summed. `Accelsearch` allows up to five stages of harmonic summing, that is, up to a maximum of sixteen harmonics.

The staged summing method is outlined in Algorithm 3 and is shown diagrammatically in Figure 3.5. This method does not give the summed harmonics of a single fundamental plane. Rather, each stage results in the summed harmonics of a different fundamental plane, each being an ever-decreasing fraction of the first plane. Represented formally: if one considers a single r-$\dot{r}$ point at $(r_1, \dot{r}_1)$, the result of the first stage of harmonic summing is simply the fundamental value at $(r_1, \dot{r}_1)$. The second stage is the sum of the first stage and the value at (½r_1, ½$\dot{r}_1$). Stage three adds (¼r_1, ¼$\dot{r}_1$) and (¾r_1 , ¾$\dot{r}_1$) to the result. This procedure continues with each stage summing twice the number of values as the previous one. This type of summing does not give the sum of one, two, and four harmonics of $(r_1, \dot{r}_1)$ but rather, the result is: the fundamental harmonic at $(r_1, \dot{r}_1)$, the sum of two harmonics of (½r_1, ½$\dot{r}_1$), and the sum of four harmonics of (¼r_1, ¼$\dot{r}_1$), etc. Since this method employs multiple fundamental planes instead of just one, the initial — and largest — fundamental plane is hereafter referred to as the *primary plane*. The collection of a primary plane and all of its sub-planes is referred to as a *family* of planes.

The principal advantage of this method is that for each stage of harmonic summing the last harmonic is the same size as the primary plane. This results in the largest fundamental plane for each stage of harmonic summing, while still being able to use the results of the previous stages.

Algorithm 3 Harmonic summing in stages

```
1:  for Each sub-plane S_w do
2:      for Each column i of the fundamental plane do
3:          Calculate x_si the index in the sub-plane
4:      for Each row j of the fundamental plane do
5:          Calculate y_si the index in the sub-plane
6:  for Each stage of harmonic summing t do
7:      for Each sub-plane S_w of the stage do
8:          for Each element of the fundamental f_(x,y) do
9:              f_(x,y) = f_(x,y) + s_(x_si,y_si)
10:     for Each element of the fundamental do
11:         if f_(x,y) is above P_thresh then
12:             Add it to the collection of candidates
```

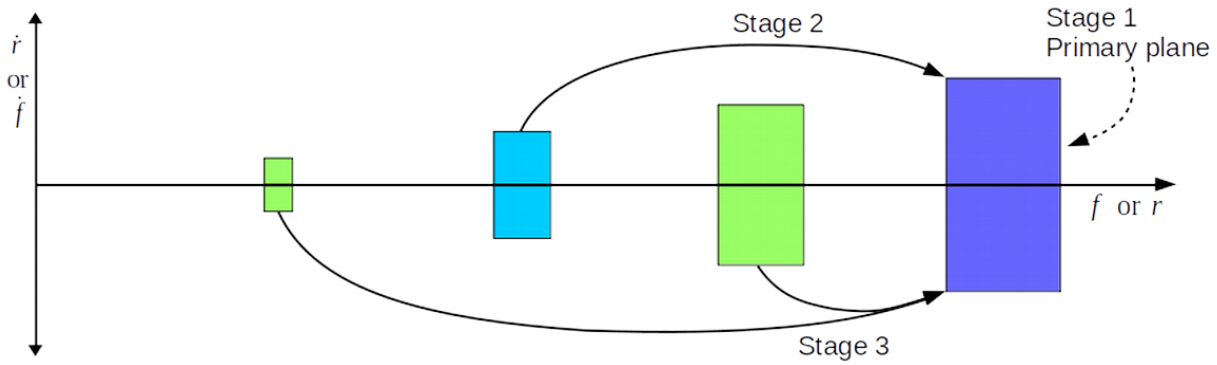


Figure 3.5: Staged harmonic summing of a family of four r-$\dot{r}$ planes. Using the staged harmonic summing approach, the primary plane is the last and largest plane, with appropriately sized sub-planes. Staged harmonic summing of such a family would be split into three stages. Stage one is simply the primary plane, stage two scales the "half" sub-plane and adds it to the primary, while stage three scales up the one- and three-quarter planes and sums them to the result. Each stage produces the summed harmonics of a different fundamental plane. After each stage, the summed powers can be searched for values above the detection threshold.

3.4.3.1 Harmonic summing in the in-memory variant

Harmonic summing in the in-memory variant uses precalculated powers stored in the full r-$\dot{r}$ plane. In the in-memory search, the full r-$\dot{r}$ plane is created incrementally by calculating small sections of the r-$\dot{r}$ plane in working memory and copying the relevant powers of each to a single large memory space. The full plane is created in increasing r, so that once a section of the r-$\dot{r}$ plane has been written to the full plane, harmonic summing can be undertaken in stages, using the last section added as the primary plane. Summing and searching are performed in a section of working memory of the same size as the primary plane. Each harmonic should have points spaced at a resolution proportional to the harmonic. However, the full r-$\dot{r}$ plane is created at a single resolution. Thus, when accessing values for harmonic summing, the closest location in the full r-$\dot{r}$ plane is used. This can be visualised as the sub-planes – which are all at the same resolution – being stretched to match the size of the primary plane. This means that the harmonics are sampled at resolutions that differ relative to the other harmonics, with the lowest harmonics having the lowest resolutions. This is another advantage of summing in stages, as having the highest harmonic at the highest resolution will give the most accurate location of a local maximum in the summed plane.

Using the full r-$\dot{r}$ plane requires accessing memory that may be widely dispersed in main memory, which can lead to some reduction in performance. However, the alternative of having each iteration calculate all the r-$\dot{r}$ values needed for that segment has a significantly higher performance cost. Thus, harmonic summing in the in-memory variant will take longer than the standard variant, but the efficiency gains in plane creation will far outweigh the losses in harmonic summing.

3.4.3.2 Harmonic summing in the standard variant

In a standard search, an entire family of r-$\dot{r}$ planes is created in each iteration. This results in some duplication of work as each sub-plane is a section of the primary plane of another segment. Each sub-plane is created at the same resolution as the primary plane. As a result, this variant has the same type of multi-resolution summed plane as the in-memory variant, meaning that they should produce very similar results.

3.4.4 Searching

For each segment, after each stage of harmonic summing, the powers of the resulting summed section of r-$\dot{r}$ plane are searched for candidates. This is done by locating values above some detection threshold. This threshold is determined from the p-value of the expected distribution of the powers of noise in the absence of any signal [130]. The power in a single frequency bin of a power spectrum, which has been normalised to unity, has an exponential distribution – a χ^2 distribution with two degrees of freedom [43]. Furthermore, the power P_h, obtained by summing h spectral bins, has a χ^2 distribution with $2h$ degrees of freedom [43]. However, this type of search covers r-$\dot{r}$ planes containing very large numbers of Fourier powers, thus the detection threshold is adjusted to compensate for N_{IFS} independent Fourier powers [115]. Combining these, the detection threshold P_{thresh} is obtained from:

$$Pr(P_h \geqslant P_{\mathrm{thresh}}) = 1 - \left(\frac{\gamma(h, P_{\mathrm{thresh}})}{\Gamma(h)} \right)^{N_{\mathrm{IFS}}} \tag{3.3}$$

In the equation above, $\gamma(h, P_{\mathrm{thresh}})$ is the lower incomplete gamma function defined as $\int_0^{P_{\mathrm{thresh}}} t^{h-1} e^{-t} dt$

and Γ is the gamma function.

In `Accelsearch`, the detection threshold is specified with a command-line parameter given as a sigma value, with a default of 2σ. Each stage of harmonic summing has to use a separate threshold, calculated from the same sigma value.

3.4.5 Candidate storage

The powers that exceed the detection threshold are flagged as potential candidates. A single periodic signal can create a large feature in the r-$\dot{r}$ plane, as is seen in Figure 2.9. These large features generally result in many localised r-$\dot{r}$ values above the detection threshold and in these cases only the location of the maximum power is stored as an initial candidate. A signal will usually result in a number of candidates at similar r-$\dot{r}$ locations but with differing numbers of harmonics. In both cases, candidates within a distance of seven Fourier bins are considered duplicates and only the best one is stored. To compare summed powers, their p-values are expressed as sigma values and these are used to rank the candidates. Calculating these sigma values can be a computationally intensive task as it is performed in double-precision and makes use of the iterative Newton-Raphson method. The initial candidates are stored in a linked list ordered by r. Adding a candidate to this list can become time consuming if the list becomes large, as each comparison requires finding a value in the list, which necessitates iterating over a large proportion of the list.

3.4.6 Candidate optimisation

The CG stage produces a large number of initial candidates, many of which may be harmonically related. For a collection of harmonically related candidates, only the fundamental should be optimised. The other initial candidates are considered "spurious" and are discarded to reduce unnecessary work during candidate optimisation. To correctly identify a collection of harmonically related candidates, it is necessary to have access to the entire collection of initial candidates. This is the reason that the CO stage is dependent on the completion of the CG stage and that candidates are not optimised during the CG stage. Once the spurious candidates have been culled, the remaining candidates are optimised.

This optimisation is performed in an iterative manner with each iteration optimising a single candidate. Optimisation consists of two basic components: *location refinement*, and *metric calculation*. Location refinement entails finding the r-$\dot{r}$ point with the highest power. This is accomplished using the Nelder–Mead method [85], to refine the location of a local maximum in the harmonically summed r-$\dot{r}$ plane. The Nelder–Mead method is a derivative-free optimisation that uses a simplex – a triangle in the case of our 2D r-$\dot{r}$ planes. This simplex iteratively "walks" around a parameter space and converges on a local maximum at ever-increasing resolutions. The location refinement is carried out as two stages of Nelder–Mead refinement, one using shorter standard-accuracy filters and the second using longer high-accuracy filters, using double-precision. The r-$\dot{r}$ values used in this process are obtained by direct evaluation of Equation 2.9.

Once the r-$\dot{r}$ location has been refined, a number of corresponding metrics are calculated. These include the incoherent power, significance, frequency second derivative, and approximated coherent power. These values describe a final candidate and are written to a binary and text file to be examined by a human operator at a later stage.

Chapter 4

General-purpose graphics processing units

GPUs are discrete graphics coprocessors which originated from the need to have a separate high-performance processor specifically to render geometry and images to a monitor. The graphics-specific requirements moulded these coprocessors to focus on computational throughput at a slight cost to latency. As GPUs have advanced, the range of operations they can perform has expanded, so that modern GPUs can be used to perform particular types of general-purpose computation, which is termed GPGPU. The computational power and cost efficiency of GPUs mean that they now play a central role in HPC. GPUs account for 36% of the computational power of the top 500 supercomputers [36] and account for 95% of the peak performance of the worlds largest supercomputer [80].

This chapter gives some basic detail on the GPGPU programming model (Section 4.2), the modern GPU microarchitectures (Section 4.3), and optimisation considerations (Section 4.4), with emphasis on Nvidia GPUs as these are used in this work. More in-depth details can be found in the CUDA C programming guide [91] and the CUDA runtime API [92]. The chapter concludes with two case studies of GPU-acceleration in pulsar astronomy (Section 4.6).

4.1 GPGPU computing

The key to the power and ultimate success of GPUs is the fact that many graphics computations are algorithmically distinct from many traditional CPU computations. Typical CPU tasks, such as running an operating system (OS) and user-level applications, are generally heterogeneous, branch-heavy and rely on some sequential progression of logic. Furthermore, these computations generally require a low latency to keep real-time systems as responsive as possible. This is in stark contrast to the traditional tasks of the graphics rendering pipeline, which encompasses the tasks required to render a 3D scene to a 2D screen. These tasks include model and camera transforms, lilting, projection, clipping, viewpoint transforms and rasterisation. Generally, these

tasks involve performing a similar transform on a large collection of polygon vertices or applying a common operation independently to a large collection of pixels. Typically, these processes are highly homogeneous, have minimal branching, and have a high degree of data parallelism. These factors helped to shape GPU hardware to focus on massively parallel and homogeneous processing, which is focused on computational throughput. To this end, the number of cores were increased and the on-chip caches were reduced, as generally, only a few threads benefit from spatially local caches. To compensate for the smaller caches, GPUs use the combination of fast context switching and massive multi-threading to hide the latency of moving small sections of data from off-chip memory to on-chip caches. This separation of computation and I/O allows some threads to perform computation while others wait for memory transactions, sacrificing single-thread performance and latency for overall throughput.

In 2007, Nvidia released CUDA, a platform specifically for performing GPGPU computation. CUDA is developed and maintained by Nvidia, and only operates on Nvidia GPUs. In 2008, the Khronos Group (a non-profit technology consortium) released Open Compute Language (OpenCL), which is an open-source alternative to CUDA. This standard open-source framework provides a programming model that allows heterogeneous parallel computing on a variety of hardware backends. These include CPUs, GPUs, and other processors from any vendor that provides an implementation. This versatility does, however, mean that OpenCL codes almost always run slower than their CUDA equivalents. The ubiquity of Nvidia GPUs in HPC and the performance of CUDA led us to use them in this research; consequently, the remainder of this work will focus on CUDA and Nvidia GPUs.

4.2 The CUDA programming model

CUDA has three primary means to accelerate an application: high-level libraries, compiler directives and language extensions. These three methods trade ease of implementation for performance. At the highest level, there are several CUDA-accelerated libraries that the developer can make use of with little knowledge of the underlying GPU architecture. The population of these libraries is growing rapidly, with CUDA Fast Fourier Transform (cuFFT) [100], cuBLAS [96], cuSPARSE [101], cuRAND [98], cuSOLVER [99], CULA [97] and AmgX [95] being a few of the libraries in common use. These libraries provide fast GPU implementations of common

mathematical operations, and can often be substituted directly into existing CPU codes.

At a level below these libraries, the OpenACC API [102] allows a developer to add compiler directives to existing C/C++ or Fortran code. This method is easy to implement and can be combined with many of the standard libraries, allowing fast and easy access to the computational power of a GPU. The third method uses extensions to standard programming languages (CUDA C, CUDA C++, CUDA Fortran). These extensions are a collection of CUDA-specific functions and language extensions that allow a developer to write CUDA-specific code in a pre-existing programming language. This is by far the most powerful of the methods, but requires the greatest understanding of the underlying architecture. In most cases, it requires a complete code restructuring and adds a large degree of complexity and associated development time. However, the flexibility and control this affords often leads to the greatest performance gains. Our work is implemented as a custom code using the CUDA C/C++ code extension and a number of the standard libraries.

A CUDA program using the language extensions is organised into two sections, a driver program that runs on the CPU (hereafter referred to as the host) and a number of CUDA functions or *kernels* that run on the GPU (hereafter referred to as the device). At the core of the CUDA programming model are three key abstractions – a hierarchy of thread groups, a hierarchy of memories, and a number of synchronisation functions. These three abstractions are discussed below, while the next section focuses on the underlying device architecture that lead to these abstractions.

The CUDA programming model and thread hierarchy have remained relatively constant since they were introduced, while the underlying architecture has evolved through seven generations. Nvidia CUDA-capable GPUs are versioned by compute capability, a two-part number where the major number indicates the generation and the minor number marks slight changes within the generation. Each generation is colloquially named after a historic scientist: Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, and Turing. Within each generation, Nvidia has three main classes of GPU: *GeForce* cards are predominantly aimed at gaming, while the *Quadro* range is designed for work stations, while the top-end *Tesla*[1] range is aimed at compute-specific servers.

[1] Some confusion may arise from this, as Nvidia also named the first generation of CUDA-capable GPUs Tesla. However, these are now mostly obsolete and not discussed in this text.

4.2.1 CUDA thread hierarchy

The first of the three abstractions is the CUDA thread hierarchy, which is divided into four levels: threads, warps, blocks and a grid (Figure 4.1). The fundamental unit of device parallelism is the CUDA *thread*. Each kernel launched by the host is executed by a number of device threads. These threads are grouped into sets of thirty-two, called *warps*. On the device, all the threads of a warp operate in lockstep. Lockstep means that all threads of a warp are simultaneously issued the same instruction, and all instructions are performed in unison by all the threads of a warp. The link between a warp and the underlying hardware is discussed in more detail in the next section.

The next level up in the hierarchy is the *thread block*, which can be thought of as a collection of warps. These can vary in size according to application requirements, but are typically created with 256 threads, and have an upper limit of 1024 threads. The various warps that make up a thread block run independently; thus there is no implicit dependence on the execution order between threads in different warps. There are, however, a number of synchronisation intrinsics that allow explicit synchronisation of all the threads of a block. These synchronisation intrinsics are CUDA-specific functions that force all the threads of a block to execute all the commands preceding the synchronisation function, before any threads can proceed with execution.

The final layer in the thread hierarchy is composed of thread blocks and is called a *grid*. All the threads of a given kernel must reside in the same grid. There is no means of synchronising the thread blocks that make up a grid, other than running separate kernels, thus thread blocks are truly computationally independent of one another.

Warp divergence

The threads of a warp execute their instructions in lockstep. When some threads of a warp have to execute different instructions, such as conditional statements, a state known as warp divergence occurs. In warp divergence, one group of threads waits idle while the other group executes its operations. Once the first path of execution is completed, the roles are reversed, and the second group waits idle while the first executes its branch of instructions. This can be thought of as all the threads of the warp running the two branches sequentially, which has a clear negative impact on performance. In the worst case, if each thread of a warp were to follow a separate execution

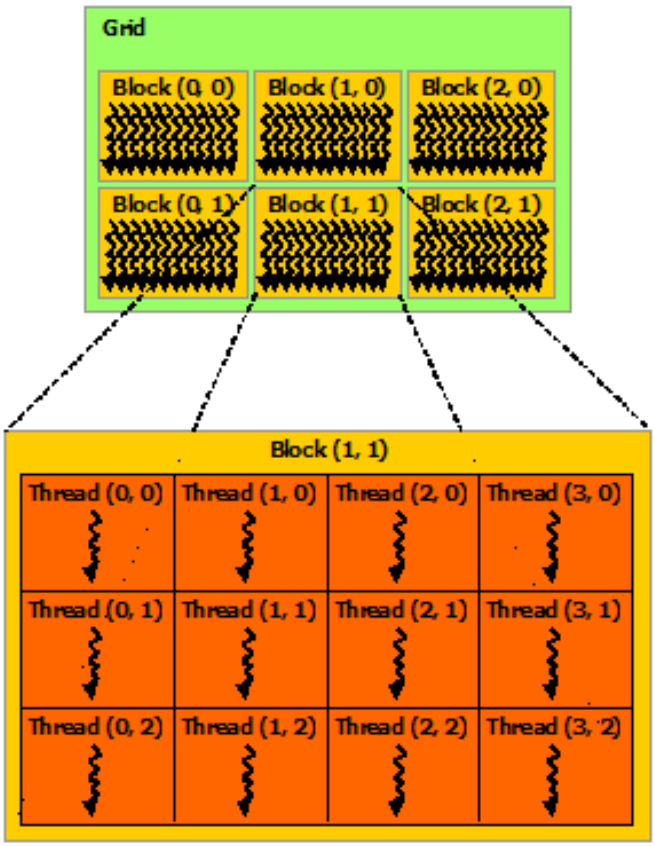


Figure 4.1: The CUDA thread hierarchy. The 2D layout of CUDA threads in which a grid is divided into a number of thread blocks, each of which contains a number of warps (a collection of thirty-two threads). Figure taken from the CUDA C programming guide [91]

path, each path would be run sequentially by all the threads, essentially running all computation sequentially [92]. Thus, warp divergence is to be minimised where possible.

4.2.2 Memory hierarchy

There are several memory spaces that a CUDA thread may access, which vary in access speed and size. The thread hierarchy is separated into three levels: global memory, shared memory, and registers. Global memory has three additional special-use memory spaces: constant, texture and surface memory. In addition, special pinned host memory can be allocated to facilitate fast memory transfers to the device. An understanding of these various memory spaces is essential, as many of the lowest level optimisations focus on memory access, and using the most applicable memory for the data concerned [91].

4.2.2.1 Registers

The fastest memory available to a device thread is the per-thread registers: these are on-chip and have the lowest latency and highest bandwidth [2]. Registers are unique to threads, although as of compute capability 3.0, threads within a warp may exchange register values using a variety of warp-level intrinsic functions known collectively as *shuffle* operations [92]. These shuffle operations are an extremely efficient means of sharing data between threads but only operate on threads within a warp. Devices of compute capability 3.0 and lower have only 63 registers per thread, while all newer GPUs have 256 registers per thread [92].

4.2.2.2 Shared memory

Shared memory is an on-chip pool of memory shared amongst all threads of a block. This memory has a low latency, but is usually very small at 48 KB per thread block. Threads within a block can share values via this memory space. This does, however, require explicit thread synchronisation between memory accesses to ensure that all the threads of the block have completed their memory transactions. Shared memory can be thought of as a user-managed L1 cache, and indeed there are configuration parameters that allow some conversion between the amounts of shared memory and L1 cache [91, §G.3.1].

4.2.2.3 Global memory

Global memory is on-device DRAM and is common to all threads. This off-chip memory has the lowest bandwidth and highest latency, with non-cached access times of 200 to 800 clock cycles, approximately 100 times slower than shared memory. All threads can access this memory; however, there is no means of explicitly synchronising threads between thread blocks. Thus, for most practical applications, this memory cannot be used to share information between the threads of a kernel. The amount of memory on a GPU can range from 2 to 16 GB, which is usually far smaller than the RAM of the host on which the device is running. Thus, the amount of memory needed is often a severe concern in GPU applications. The Quadro and Tesla ranges of GPUs come with error-correcting code (ECC) memory, which is specifically included to reduce errors

[2] Here, bandwidth is used in the computing context to mean the rate of data transfer across a given path and will be expressed in GB/s. This is not to be confused with the signal processing use of bandwidth used in the previous chapter.

in the compute-specific environments [92].

4.2.2.4 Local memory

Each thread also has thread-specific local memory. This memory is located on the off-chip DRAM and thus has similar access latency to global memory; it is therefore not particularly efficient. The compiler automatically uses this memory under a number of conditions. These include arrays where indexing cannot be determined at compile-time, and large structures or arrays that would consume too much register space [92]. Global memory can also be used for any variable if a kernel uses more registers than are available; this is known as *register spilling*. Due to the high latency of global memory accesses, register spilling can significantly reduce the performance of a kernel and is to be avoided wherever possible [91].

4.2.2.5 Constant memory

Constant memory is a special read-only section of global memory which is optimised for cases in which all the threads of a warp read the same memory location. In these cases, the value can be broadcast to all the threads of a warp in a single clock cycle. This memory is located on-device, however, it has its own on-chip cache which means that it can be very efficient if only small sections of constant memory are used.

4.2.2.6 Texture and surface memory

Texture and surface memories are global memory with a special on-chip cache. The texture cache is optimised for 2D spatial locality data. This means that if threads of the same warp read texture or surface addresses, and those addresses are close together in two dimensions, a performance gain may occur. The use of texture memory can make a large impact on very old (pre-Fermi) cards as they do not have L1 caches. However, in most modern GPUs, the use of texture memory has little impact on performance [92].

4.2.2.7 Pinned host memory

The only device memory that the host can access is the global memory, thus this memory is used to transfer data to and from the device. This is usually done by copying relevant sections of memory between device global memory and host-side RAM. The section of host RAM used is often allocated as non-pageable RAM, as this can improve transfer speeds. Non-pageable RAM will never be paged to virtual memory which is stored on slower storage. This type of memory is referred to as *pinned memory*. In many instances, two *paired* sections of memory are allocated – one is a section of device global memory and the other an identically sized section of pinned host memory. A kernel's input and output values are written to the relevant memory by either the host or the device and mirrored on the other with a single efficient memory copy. These copies are performed using the PCI Express bus or the newly introduced NVLink bus, and have bandwidths ranging from 16 to 150 GB/s [89]. This data transfer has by far the lowest bandwidth, thus copying data to and from the device can often be a bottleneck in GPGPU computation [91].

4.2.3 CUFFT library

FFTs play a large role in this work, hence some special attention is given to the CUDA Fast Fourier Transform (cuFFT) library [100], which provides a number of GPU-accelerated FFT implementations. This library consists of two separate libraries: cuFFT and cuFFTW. The cuFFT library [100] is designed to provide high performance FFTs on Nvidia GPUs, while the cuFFTW library [100] is provided as a porting tool to enable users of Fastest Fourier Transform in the West (FFTW) [41] to start using Nvidia GPUs with minimal effort.

The cuFFT library [100] provides all the basic features that may be expected: these include forward and reverse transforms in both single-, double- and half-precision. These transforms can be performed in three ways: complex input to complex output (C2C), real input to complex output (R2C), and symmetric complex input to real output (C2R). All three can be performed as in-place and out-of-place transforms.

The configuration method used by cuFFT is called a *plan* and is modelled on the highly successful and ubiquitous FFTW library [41]. A plan is an execution scheme that uses the internal building blocks to create an optimal execution plan given the user configuration and the particular GPU

hardware selected. When the execution function is called, it takes the plan as an input and – following the specific execution path of the plan, and using the input and output memory locations provided – performs the FFT. The advantage of this approach is that once the user creates a plan, the library retains whatever state is needed to execute the plan multiple times without recalculation of the configuration. For a transform of N values, this configuration calculation requires calculating $\mathcal{O}(N)$ transcendental functions, which can be much slower than the $\mathcal{O}(N \log_2(N))$ simple arithmetic operations inside the FFT computation itself. Thus, creating a plan can take significantly longer than executing it. However, using this model, the computationally expensive plan creation need only be performed once. This is done prior to a performance-critical loop in which the transform is performed multiple times. This model is effective in the GPU context because different kinds of FFTs require different thread configurations and GPU resources, and the plan interface provides a simple way of reusing configurations. When a plan is created, an associated section of working memory is allocated in GPU global memory. This memory is used to store intermediate values during the execution of the plan. The size of the work area varies and can be calculated using an API function. However, it has been observed that it is almost always the same size as the output data. The cuFFT API is thread safe, and multiple host threads can use a single plan. However, due to the plan-specific working memory, a single plan cannot be run concurrently. The synchronisation of plan execution can be controlled using the standard CUDA streams and events (Section 4.3.7).

There are some scenarios where many similar FFTs need to be performed on differing input data. With cuFFT, these many similar transforms can be grouped together into what is called a batch. All the transforms can then be performed at one time, allowing increased efficiency. This requires making a batch-specific plan, which will usually have a work area equivalent in size to the entire output of the batch. Thus, batched FFTs trade a higher memory usage for improved performance.

Version 6.5 of CUDA introduced cuFFT callback routines. Callback routines are user-supplied CUDA functions that cuFFT will call to load the input or store the FFT output to memory. These allow the user to do some pre- or post-processing of the relevant input or output data, without the need for additional kernel launches. One common use of callbacks is to reduce the amount of data read from or written to memory, either by selective filtering or via type conversions. In the callbacks, a CUDA thread is assigned to each input or output value; each thread runs the relevant

callback function before or after the transform. In a load callback, the parameters passed to each thread are the memory address of the first input value and the point-specific offset. The task of the load callback is to read from memory or generate a single input value for the FFT. Similarly, in a store callback, each thread is passed a single output value, the memory address of the first storage location, and the point-specific offset. The store callback then processes and writes the output to memory.

4.3 Modern GPU microarchitectures

In order to leverage the full performance of a GPU, some understanding of the underlying architecture is beneficial. A GPU card has two main components – memory and one or more chips. The memory is analogous to the RAM in a standard PC and makes up the global memory in the CUDA memory hierarchy, while the chip is analogous to the processor of a PC. There are two primary considerations when performing GPGPU computation: memory bandwidth and computational throughput.

In a kernel, if the amount of computation per unit of data is low, the computation may not match the required memory accesses. In these cases, the factor limiting the kernel execution time is the bandwidth to global memory; these are referred to as *memory-bound* problems. If, however, the limiting factor is the amount of computation, a kernel is referred to as *compute-bound*. With these kernels, the metric of interest is the effective computational throughput, which is usually measured in giga-floating-point operations per second (GFLOPs).

4.3.1 Bandwidth

A GPU processor accesses device memory via a custom bus with bandwidths ranging from 96 to 900 GB/s [94], this maximum hardware bandwidth is known as the *theoretical bandwidth*. The optimisation focus of memory-bound kernels should be to get the observed or *effective bandwidth* as close to the theoretical maximum as possible. The per-warp global memory access pattern plays a large role in determining bandwidth utilisation. Similar to instructions, memory transactions are issued per warp. The ideal memory access pattern is to have aligned and coalesced memory accesses [91]. Aligned memory accesses occur when the first address of a device memory transaction is a multiple of the cache segment size. Coalesced transactions occur when all the

threads of a warp access a contiguous block of memory words. If values are used multiple times, shared memory can often be employed as a user-managed cache, to reduce memory latency.

Shared memory is arranged into banks where successive words are mapped to successive banks. All current GPUs have thirty-two banks, and each bank has a bandwidth of 64 bits per clock cycle. For the threads of a single warp, a shared memory transaction can be thought of as a single transaction. However, if multiple threads access the same bank, a bank conflict occurs, and the accesses are serialised. Managing bank conflicts can have a significant impact on performance, and is often the focus of low-level optimisation.

4.3.2 Streaming multiprocessor

Figure 4.2: A diagrammatic representation of the sixth-generation Pascal GP100 SM unit. This SM has two subpartitions, each with a separate warp scheduler, instruction buffer and register file. The SM has a 64 KB shared memory and texture/L1 cache, common to both subpartitions. The SM has 64 FP32 CUDA cores, 24 FP64 CUDA cores and 16 special function units (SFUs). Figure taken from [93].

A GPU usually has a single chip, with some of the high-end compute cards having two, or occasionally, four chips. The key component of modern GPU chips is the streaming multiprocessor (SM). These SMs are the base units on which the CUDA kernels are run and contain the actual "cores" that do the computation. Most chips consist of several SMs packed together with some memory and memory controllers all on a single die, allowing chips to be built in a roughly modular fashion.

Figure 4.2 shows a diagram of the GP100 SM; this is the SM that is found in the most recent chips of the Pascal generation of compute GPUs [93]. This diagram is used to discuss a number of the key features of an SM. Table B.1 shows some these features across the SMs of a wide range of Nvidia GPUs.

The GP100 SM is divided into two subpartitions, each containing thirty-two CUDA cores. A CUDA core, sometimes referred to as a stream processor (SP), is a pipelined 32-bit floating-point/integer (FP32) execution unit, usually capable of performing one basic 32-bit operation per clock cycle. In addition, a subpartition contains an instruction scheduler and a number of 32-bit registers. When a CUDA kernel is launched, each thread block is assigned to an SM. On execution, the resource manager allocates all thread block level resources, such as shared memory. Thereafter, sufficient warps for all threads in the thread block are generated, and warp-specific resources, such as registers, are allocated. Each warp is assigned to, and executed on, a subpartition. A subpartition can concurrently run several warps in round-robin-style execution, provided it has sufficient resources for all the warps. Indeed, as discussed presently, it is desirable to have a number of warps concurrently resident in each subpartition. Instructions are issued on a per-warp basis, with all the CUDA cores of a subpartition performing the same operation but with unique registers. This results in the lockstep execution of threads in a warp. Once all the warps of a thread block are complete, the block-level resources are released and the compute work distributor is notified that the block has completed execution. Due to the fact that on a hardware level, manual synchronisation is only possible between threads within a single thread block, synchronisation is only possible within an SM. Thus, separate SMs operate completely independently. This modularity allows the parallel architecture to scale.

The term CUDA core refers to a single FP32 execution unit, however it is often noted that calling it a core may be an overstatement. Rather, the title of a "core" would be more accurately attributed

to a subpartition, as it has a single instruction scheduler and instruction buffer. However, this would be a core capable of performing thirty-two identical arithmetic operations per clock cycle. This work will, however, keep to the convention of referring to a single FP32 execution unit as a CUDA core.

4.3.3 Occupancy

Instruction throughput is the number of clock cycles it takes for an instruction to complete execution. A number of factors influence the instruction throughput that can be attained on an individual SM. The *peak throughput* is determined by the number of instructions that can be issued per clock cycle. In the case of an SM, instructions are issued by the warp scheduler. In Table B.1 one can see that for the generations of interest (Kepler and later) this is either two or four. This peak throughput is not always attained, as there is an architecture-dependent *execution latency* which is six clock cycles for Maxwell and Pascal. This latency is usually a result of register dependency, and can be mitigated by queuing instructions in the *instruction buffer*. These instructions are usually from separate warps. Thus, the per-SM peak throughput can only be attained when the number of instructions, and thus active warps, exceeds the product of the peak throughput and the execution latency. For example, Maxwell generation GPUs need at least $24\ (6 \times 4)$ active warps per SM to attain peak throughput. This gives a rough minimum number of warps per SM of 18, 36, 24 and 12 warps per SM for Fermi, Kepler, Maxwell and Pascal respectively. These minimums assume only basic 32-bit instructions using operands in registers. If there is some instruction-level parallelism (ILP) present, this can effectively reduce the execution latency, in turn reducing the minimums above. Thus, as a general rule, it is desirable to have many warps per SM, as this is more likely to maximise instruction throughput.

The number of concurrent thread blocks, and thus warps, in an SM depends on the resource requirements of the warps. For instance, the quantity of shared memory that is needed per thread block, and how many registers are used by each warp, can limit the number of warps that can be run on an SM. The metric used to measure the number of active warps is *occupancy*, which is the ratio of active warps to the maximum possible number of warps on an SM. The maximum number of warps is architecture-dependent: 48 for Fermi, and 64 for all later generations apart from compute capability 3.7, which has a maximum of 128 warps. Thus, the minimum occupancy

to cover basic execution latency is calculated as the minimum number of warps required for peak throughput, divided by the maximum number of warps on an SM. In the case of Maxwell GPUs this is 37.5% (24/64).

4.3.4 Latency hiding

The occupancy calculated above assumes that the only operations being performed are basic 32-bit operations that use only registers. In reality, most kernels issue some instructions that have a far higher latency, in particular memory loads, which can have a latency of up to 1000 cycles. Due to the relatively small caches, many memory operations require accessing uncached data from global memory. The general GPU approach is to have many other warps that can run while these long latency operations are undertaken, called *latency hiding* as it allows a high throughput. Accordingly, there is a strong drive when optimising a CUDA kernel to increase occupancy. However, it is important to note that the true objective is not occupancy but throughput. A kernel with low resource requirements that requires many values from global memory may allow a high occupancy. However, much of the time may be spent stalled, waiting for memory transactions, reducing actual throughput. Thus, the true aim of latency hiding is to efficiently balance compute-heavy tasks with memory access. Therefore, in some cases it is better to perform some "redundant" calculations if this avoids some memory accesses.

4.3.5 Precision

The CUDA cores discussed previously perform only 32-bit operations. Double-precision (FP64) execution units are specifically designed to perform double-precision calculations. It is noted that in the GP100 SM, there are half as many FP64 cores as there are FP32 cores (Figure 4.2). Thus, this SM can perform half the number of double-precision as single-precision floating-point operations per second (FLOPs). This 2:1 ratio is common with the top-end compute cards, with some going down to 3:1 or 4:1. However, this ratio of FP64 cores has specifically been increased for these computation-specific cards. In the traditional realm of computer graphics, GPUs are extremely biased towards single-precision calculations to the extent that most of the SMs in consumer-level gaming GPUs have only a single FP64 core for an entire subpartition, and therefore the ratio is 32:1. Thus, GPU computation and many GPU codes are still heavily bi-

ased towards single-precision calculations as there may be architectures where double-precision calculations take thirty-two times longer to perform than single-precision equivalents. This is demonstrated in Table C.1 where it is evident that the double-precision trigonometric functions take hundreds of clock cycles to compute, whereas the single-precision equivalents take only tens of clock cycles. The ratio of clock cycles for equivalent functions is very close to the expected 1:32 for GeForce GPUs.

As of CUDA 7.5, Nvidia introduced the half-precision floating-point number (FP16) as a storage type. These 16-bit floating-point numbers are in the standard Institute of Electrical and Electronics Engineers (IEEE) 754-2008 binary2 format [91] and were introduced as a storage type only. This means that to perform a calculation using these values requires that they first be converted to a 32-bit floating-point value. Where full 32-bit precision is not needed, these 16-bit values can reduce storage, register use and bandwidth requirements. This can significantly offset the minimal computation required for conversion. In recent years, there has been a drive to use GPUs in machine-learning applications, specifically deep-learning neural networks. These neural networks require lower accuracy as they have natural fault tolerance due to the backpropagation algorithm used in their training [25]. This drove the need for FP16 calculations, and since compute capability 5.3 GPUs include the capability of performing 16-bit floating-point arithmetic [91]. These operations are done as paired computations using the existing FP32 cores [93], effectively doubling the half-precision FLOPs in the latest generation of Nvidia GPUs.

4.3.6 Instructions

4.3.6.1 Special function units

In addition to the computational units already discussed, an SM contains a number of SFUs. These are processing units capable of computing a number of transcendental instructions such as basic trigonometric functions, reciprocals, square roots, and graphics interpolation instructions. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue other instructions while the SFU operates [88]. The SFU instructions are usually performed in single-precision, with many of the instructions taking one clock cycle to complete. In the GP100 SM, there are four FP32 cores for each SFU: this 4:1 ratio is very common across most newer SMs, with an 8:1 and 6:1 ratio in Fermi and Kepler generations respectively. Table B.1 gives a sample

of the ratio of FP32 cores to SFUs for a range of GPUs.

4.3.6.2 Intrinsic functions

The CUDA API exposes a number of basic functions; some of these come in two variants – standard and *intrinsic* functions. The standard functions can be used in both host and device code, and the majority of the basic floating-point functions, such as multiplication and addition, are IEEE-compliant. These functions usually come in single-, double-, and now half-precision, and often map to a basic instruction of an FP32, FP64 or FP16 core. Intrinsic functions can only be used in device code; they are less accurate but faster versions of some of the standard transcendental functions. They have the same name as the standard functions but are prefixed with `__`, such as `__sinf(x)`. Many of the intrinsic functions map to an instruction of the SFU. Thus, many single-precision intrinsic functions such as `__sinf(x)` can be performed by an SFU in a single clock cycle. However, due to the standard 4:1 ratio, it can take four cycles for an entire warp to complete a `__sinf(x)` calculation. However, as the operations are performed using the SFUs, the SM can continue with other calculations on the standard CUDA cores. This additional ILP allows a higher throughput when using intrinsic functions. The SFU operates at single-precision, thus there are many cases where there are no double-precision intrinsics, such as the trigonometric functions.

Table C.1 shows the actual number of clock cycles taken to complete some basic device functions. It can be seen that the number of clock cycles required to perform the intrinsic `__sinf(x)` function match the ratio of FP32 cores to SFUs (Table B.1) for the three GPUs used. This indicates that on these GPUs it takes one clock cycle for an SFU to calculate the `__sinf(x)` function and similarly, two clock cycles to calculate the `__sqrtf(x)` function.

4.3.6.3 Atomic operations

There are some *atomic operations* which can operate on both shared and global memory. Atomic operations are basic read-modify-write operations, such as addition or multiplication, which operate on a single word in memory. These operations are guaranteed to be performed without interference from any other thread and thus allow some asynchronous – yet safe – operations on both shared and global memory. Any concurrent atomic operations on the same memory location

will be serialised. In the worst case, if all the threads of warps access the same memory address at the same time, all operations will be run sequentially. Atomic operations allow a greater range of functionality, however, they can be relatively slow, especially on older hardware (compute capability < 3.0).

4.3.6.4 Timing

Table C.1 shows the run times of a number of GPU functions, including a number of intrinsic functions. One can see from this table that many of the intrinsic functions run in four clock cycles as would be expected with eight SFUs per subpartition. The GPUs used for these timings have only one double-precision unit per subpartition. Accordingly, even the basic double-precision operations, such as addition and multiplication, take approximately 32 clock cycles, while the more complex functions such as the double-precision trigonometric functions can take hundreds clock cycles to complete. Thus, the accuracy of double-precision can come at the cost of increasing the run time by approximately 100 times when compared to the single-precision intrinsics. This highlights why many GPU codes, including those in our research, use as many single-precision calculations as possible.

4.3.7 Asynchronous execution

In the most basic generic GPU computation model, the host prepares some data in the host memory, transfers this to the GPU, and a CUDA kernel is launched to perform some computation on the GPU – the outputs of which are stored in the device memory. After the kernel has completed its execution, the results are copied from device memory to host memory, after which the CPU can perform some action with the result. In this model, all components wait until the previous component is complete before being able to execute, and the CPU is essentially idle while the memory transfers and GPU computation occur. This type of execution is referred to as *blocking*, since each GPU operation blocks the CPU execution. Most CUDA devices of compute capability 2.0 and higher can perform both asynchronous kernel launches and asynchronous data transfer to and from the device [91]. These asynchronous launches allow a host thread to call a GPU function and then continue execution while the memory transfers or GPU computation is performed in the background.

Synchronisation of these asynchronous components is performed using CUDA *streams* and *events*. A CUDA stream is an execution pipeline; kernels and memory copies which are launched asynchronously into the same stream are guaranteed to execute consecutively in the order that they were launched. However, kernels launched into different streams are permitted to execute concurrently. CUDA events are checkpoints which can be created asynchronously in a specific stream and are used to mark the execution boundaries of components within a stream. The CUDA API has a number of functions that allow synchronisation of the execution in a stream, dependent on the state of an event. Thus, events are used to facilitate synchronisation between CUDA streams. Events can be time-stamped, this enables accurate post-execution timing of CUDA kernels and memory transfers.

Using these streams and events, it is possible to launch *non-blocking* memory copies and kernels, and manually manage synchronisation. Memory copies to the device may be launched in separate streams and will adhere to the execution order of their various streams. However, it is important to note that they will all run serially with respect to each other, similarly for memory copies from the device [91]. The majority of compute-specific GPUs allow copying in both directions at one time, while on many of the GeForce range of cards, all memory transfers are run serially in the order order in which they were launched. These memory copies can still be performed completely asynchronously with respect to all GPU computation, which enables hiding the latency of copying data to and from the device. In addition, CUDA allows a number of kernels to run simultaneously on a single device, and indeed warps from multiple kernels can be run on a single SM. If these are all combined, up to four tasks can be performed concurrently, namely: executing multiple CUDA kernels; performing memory copies to the device, performing memory copies from the device, and CPU computation. When all four are achieved, this is known as *four-way concurrency*. Naturally, care needs to be taken not to copy, read or write device or host memory at the wrong time, increasing the complexity of manual synchronisation.

4.4 Performance guidelines

The CUDA Programming Guide [91] provides several basic optimisation strategies that take into account many of the points discussed in this chapter. This section gives an outline of the three main considerations; the first of these is to maximise parallel execution to achieve maximum

utilisation. The second is to optimise memory usage to achieve maximum memory throughput, while the third is to optimise instruction usage to achieve maximum instruction throughput.

4.4.1 Utilisation

To maximise utilisation, the application should be structured in such a way that it exposes as much parallelism as possible. This should be efficiently mapped to the various components of the system to keep them busy most of the time. At a high level, the application should maximise parallel execution between the host, the devices and the bus connecting the host to the devices, by using asynchronous function calls and streams. The application should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

At a lower level, the application should maximise parallel execution between the SMs of a device. This can be done by aggregating many similar smaller tasks or kernels into a single larger kernel that may allow more well-sized thread blocks to be created. In addition, multiple kernels can be executed concurrently on a device, ensuring that there are always new thread blocks to be run on an SM when its resources become free.

At an even lower level, the application should maximise parallel execution between the various functional units within an SM. SMs rely on thread-level parallelism to maximise utilisation of their functional units. Thus, utilisation is directly linked to the number of warps resident on an SM. For a given warp, there is some latency for each instruction called. To hide this latency, there needs to be sufficient resident and idle warps that a pipeline of instructions can be formed. If some input operand resides in off-chip memory, the latency is much higher: 200 to 800 clock cycles. The number of warps required to keep the warp schedulers busy during such high latency periods depends on the kernel code and its degree of ILP. The number of resident warps is limited by the availability of shared memory and registers. Thus, the use of shared memory and registers can have a significant impact on the number of resident warps and, therefore, performance.

4.4.2 Data throughput

The first step in maximising overall memory throughput for the application is to minimise data transfers with low bandwidth. The lowest bandwidth is between the device and the host, thus care

must be taken to efficiently transfer as little data between the device and host as possible. Above that is the bandwidth between the chip and global memory; in some instances this can be reduced by using shared memory, while in others the traditional hardware-managed cache can be more appropriate to exploit data locality. Wherever possible, memory transactions to global memory should be aligned and coalesced. As a golden rule, the more scattered the memory addresses are, the lower the throughput will be.

4.4.3 Instruction throughput

Instruction throughput can be reduced in a number of ways – firstly, by trading precision for speed when it does not affect the end result, such as using intrinsics instead of regular functions or single-precision instead of double-precision. Secondly, by minimising warp divergence caused by control flow instructions. Lastly, by reducing the number of instructions, for example, by optimising out synchronisation points and minimising type conversions where possible.

4.5 Hardware

The generation of the GPU used makes a difference to a number of our CUDA kernels. Shuffle instructions (Section 4.2.2.1), were introduced from the Kepler generation onward, are used in a number of the kernels. These make a significant difference and are preferable to the alternative. Thus, a GPU of CC 3.0 or greater is recommended for use with this search.

In our testing, three GPUs with similar specifications were used – one each from the Kepler, Maxwell and Pascal generations. The GPUs used were the Nvidia GeForce GTX 770, GTX 970 and GTX 1070, the second-best GPUs in the GeForce range for each generation. These GPUs have a higher clock rate but fewer SMs than the compute-specific GPUs in their generation, thus they have lower maximum computational throughput and memory bandwidth. However, on a price-per-FLOPs basis, the GeForce GPUs are five to ten times more affordable than compute-specific GPUs. The only significant drawback of these GPUs is that – being aimed at the gaming market – they have only a single FP64 unit per subpartition, thus their double-precision computations are approximately thirty-two times slower than their single-precision computations. However, in our implementations we have been able to limit double-precision operations to a bare minimum. The GTX 1070 is used for the majority of results reported in this work; the results

Table 4.1: Details of GPUs used in testing.

Card	FP32 Cores	FP64 Cores	Mem GB	Mem Clock	Bus Bits	Bandwidth GB/s	SMs	CC	Clock MHz	FP32 Tflops	FP64 Tflops
GTX 770	1536	48	2	7010	256	224.32	8	3.0	1137	3.49	0.11
GTX 970	1664	52	4/(3.5)	7010	256	224.32	13	5.2	1190	3.96	0.12
GTX 1070	1920	60	8/(3.5)	8008	256	256.26	15	6.2	1506	5.78	0.18

of the GTX 970 and GTX 770 are only given when there is a notable difference between the generations.

The technical specifications of these three GPUs can be found in Table 4.1. Special note must be made of the fact that the GTX 970 has 4 GB of memory, however this is segmented into a 3.5 GB and a 0.5 GB section with the latter section sharing a bus used by the former section [90]. Using this last 0.5 GB can have a significant impact on effective bandwidth, thus in all testing, GPU memory use was limited to 3.5 GB on all GPUs.

The PC in which the GPUs were run was used for the CPU benchmarks. This PC was chosen as it is in a similar price range and generation as the GPUs. It has a four core Intel® Core™ i7-2600 processor, running at a clock rate of 3.4 GHz with 16 GB of DDR3 memory at a clock rate of 1067 MHz.

In this work, desktop GPUs were used for development, testing, and timing; this was done primarily due to their availability and affordability. However, a large-scale pulsar survey such as those proposed using the SKA would almost certainly be performed with compute-specific GPUs rather than desktop GPUs. The former may cost more per-FLOPs, however they have better reliability, double precision performance and more device memory. These factors make compute-specific GPUs more suitable for large scale surveys.

4.6 GPU acceleration case studies

This section presents two detailed case studies in which GPU acceleration has been used in pulsar astronomy. This is intended to give the reader an understanding of the types of GPU optimisations that are effective and are commonly used, while at the same time giving a bit more detail on the components of the pulsar search pipeline.

4.6.1 GPU acceleration case study: Dedispersion

In the classic pulsar search pipeline, incoherent dedispersion is the DSP component that precedes the periodicity search. There are three main implementations of incoherent dedispersion. The first, and simplest, is *direct dedispersion* which is described in Section 2.3.2. This method operates on a channelised time series. For each DM trial, each channel is shifted by the applicable time delay, after which all the channels are summed across frequency to form a single time series. This requires $\mathcal{O}\left(N_t N_v N_{\mathrm{DM}}\right)$ floating-point operations, where N_v is the number of frequency channels, N_t is the number of time samples, and N_{DM} is the number of DM trials.

The second method, *sub-band dedispersion*, uses a hierarchical approximation whereby the full frequency band is divided into a number of sub-bands [77]. Dedispersion is then performed in two stages. The first stage performs a subset of the DM trials, as a direct dedispersion of the channels in each sub-band. The results of this are then used in a second round of direct dedispersion that takes the dedispersed sub-bands as input and computes the full range of DM trials. Thus, the first step operates on a reduced number of DM trials while the second operates at a reduced frequency resolution, thereby reducing the overall computational cost. The second stage of this method uses wider frequency bands; these have been dedispersed in the first stage, however, the wider bands do still increase the error introduced by smearing. Thus, this method trades some accuracy for computational efficiency.

The third alternative, *tree dedispersion* – first described by Taylor [125], arranges the computation so that repeated calculations can be shared among different DM trials. This allows the computation to be performed in a divide-and-conquer approach similar to the FFT algorithm. The theoretical complexity of this method is $\mathcal{O}\left(N_t N_v \log_2(N_{\mathrm{DM}})\right)$ operations, which is significantly better than the direct method. This technique, however, requires the number of channels to be a power of two. In addition, this assumes that time delays are linearly related to frequency, whereas in reality, this relationship is quadratic. The last violation can result in a significant reduction in accuracy. The standard solution is to approximate the quadratic relation as a number of piecewise linear segments. As with the second method, this requires dividing the frequency range into a number of sub-bands. For every DM trial, each sub-band is dedispersed using the tree dedispersion method, after which all the sub-bands can be summed across frequency to give the final time series. In each sub-band, the quadratic dispersion is approximated by a linear re-

lationship which introduces some smearing. This smearing can be minimised by increasing the number of sub-bands, however this, in turn, decreases the computational efficiency. Therefore, this method has a similar trade-off between accuracy and efficiency. The theoretical computational complexity of this method is lower than the others; however, implementing it is sufficiently complicated that it is not used in many real-world pulsar search pipelines.

4.6.1.1 GPU acceleration of dedispersion

There are a number of GPU-accelerated implementations of dedispersion, these include work by Magro et al. [73], Armour et al. [4], De and Gupta [27] and Barsdell et al. [12]. In this section, we examine only the work of Barsdell et al., who wrote and timed GPU implementations of each of the three methods discussed above. These were intended for research purposes, and as such, none of the implementations were fully functional. A number of the final timing results were extrapolated from the timing of individual components and assumptions on scaling. These results do, however, give a good indication of the run times and speed-up attainable through incoherent dedispersion when run on a GPU. It was found that tree dedispersion – although the most efficient on paper – performed worse than the sub-band method with respect to computation time. This was attributed to the increased complexity of the method, combined with a higher memory requirement, both of which may hamper GPU computation. The efficiency of the tree method was poor, but the smearing error in the sub-band method was significantly worse than that of the tree method – by up to three orders of magnitude [12]. This clearly shows the trade-off between accuracy and computation: the slowest (direct dedispersion) is the most accurate, while the fastest (sub-band dedispersion) has the greatest error. Tree dedispersion has a lower error than the sub-band dedispersion but requires longer run times.

Barsdell et al. performed timing tests using one minute of data on a 1024 channel filter bank sampled at $64\,\mu s$. The GPU run times were significantly faster (10–60 X) than their optimised CPU direct dedispersion equivalents, run using four CPU cores. With their CPU direct dedispersion implementation, Barsdell et al. were able to perform approximately 684 dedispersions in the duration of the observation. Similarly, using a GTX 480 with their recommended sub-band sizes, they were able to perform 6248, 43322, and 19393 dedispersions with the direct, sub-band, and tree dedispersion methods respectively. Considering that most common blind searches have

1000 to 10000 DM trials, using even a single GPU, the dedispersion component can be run in real-time. The dedispersion speeds obtained by Barsdell et al. [12] are better than real-time, thus there is scope to increase the number of frequency channels and still run in real-time. This is an example of the kind of accelerated computation that this work endeavours to achieve.

4.6.2 GPU acceleration case study: Acceleration search

Concurrent to this work, Dimoudi et al. have begun developing a GPU-accelerated implementation of the FDAS. Their implementation is intended to be part of the `AstroAccelerate` project [30] which will be incorporated into the SKA pulsar search backend. The vast amount of data that will be created by the SKA necessitates real-time processing of the search data, as the raw data cannot be stored for an extended period [66]. Thus, the main requirement of their implementation is that it is capable of running in real time. The initial results of their GPU-accelerated implementation of the FDAS are given in Dimoudi et al. [30].

Here, we examine the proposed SKA surveys to determine the requirements for real-time searching. In the proposed pulsar searches on the SKA1-Mid telescope, each approximately nine-minute pointing will have 1500 beams [66]. The periodicity search will perform up to 100 acceleration trials on at least 500 DM-corrected time series [66]. These searches will be conducted on a cluster of 500 compute nodes, each containing two accelerators, which are currently proposed to be one GPU and one field-programmable gate array (FPGA) board [66]. Each of these nodes will be required to process three beams in real time [66] (multiplied by 500 DM trials, this results in 1500 time series per node). This processing includes beamforming, dedispersing, periodicity searching and folding [66]. Thus, for a real-time periodicity search, each node would be required to perform a 100-trial acceleration search on 1500 dedispersed time series of 8 million samples each.

Dimoudi et al. have implemented a GPU-accelerated pipeline that is capable of creating a single large r-$\dot{r}$ plane using the overlap-and-save technique. This is analogous to creating the in-memory plane in `Accelsearch` (Section 3.1). Currently, their pipeline only creates the full r-$\dot{r}$ plane, and does not perform harmonic summing, searching or CO. Thus, the only components it performs are those of the plane creation task (Section 3.4.2), excluding segment input normalisation. It is important to note that their implementation differs from `Accelsearch` in the way that

the frequency resolution is increased. They perform interbinning on the acceleration-corrected Fourier components as opposed to `Accelsearch` which uses fine binning to achieve the same frequency resolution of 0.5 bins. The former is less computationally intensive as it sums Fourier components post-convolution, whereas the latter increases the filter resolution, which in turn increases the size of the FFT and iFFT used in the convolution.

In Dimoudi et al. [30] the results of two alternative implementations are presented and compared. The first uses cuFFT to perform the various FTs, while the second makes use of a custom FFT implementation, which incorporates the complex multiplication and power calculations. They show that the latter is approximately 1.5 to 2.5 times faster than the former. What follows is a detailed analysis of the first method; thereafter, the second method is discussed to reveal the means by which speed-ups are achieved.

The first implementation performs the initial real-to-complex FFT of the full data set using cuFFT. The segments are then FFTed using a batched complex-to-complex cuFFT transform. Each transformed segment is then pointwise multiplied with each of N_Z acceleration templates using a custom CUDA kernel. The results are then iFFTed using a batched complex-to-complex cuFFT transform. Finally, a custom CUDA kernel is used to compute the Fourier powers for each r-$\dot{r}$ plane and store the uncontaminated values to the full in-memory r-$\dot{r}$ plane.

Two alternative multiplication kernels were investigated. The first uses a one-dimensional grid, with one thread per column, while the second uses a 2D grid with one thread per r-$\dot{r}$ point. The authors note that both alternatives are memory bandwidth limited due to the low arithmetic intensity of the pointwise multiplication, and thus have similar run times. The first alternative is preferred, although it has a high register usage as each thread loads a full column of the convolution kernel into registers. This kernel can thus suffer from low occupancy and register spilling.

In this first implementation, the multiplication component comprises approximately 20% of total run time. The cuFFT transforms account for around 42% of run time, while the kernel used to compute the powers accounts for 35% of run time. This implementation is bandwidth limited, as each component has to read and write intermediate results from global memory. These high-latency memory transactions can hamper the performance of the convolution.

The second implementation created by Dimoudi et al. was developed to compensate for the band-

width limitations when using cuFFT. This uses a custom FFT code which reduces the reads and writes to global memory by incorporating the multiplication and the power calculations into a single CUDA kernel. The kernel uses shared memory and registers to store intermediate values, reducing the memory transactions to global memory. Due to the reliance on shared memory, their current implementation is limited to a maximum filter width of 1024 points ($\omega = 1024$). For the forward DFT they use the decimation in frequency (DIF) variant of the Pease algorithm [106], while for the inverse DFT they use the decimation in time (DIT) variant of the Cooley-Tukey algorithm [24]. Using a DIF and DIT algorithm for the two directions of the DFT allows them to eliminate two re-ordering steps, reducing the number of operations performed and thus the execution time. Their custom FFT implementation is optimised to make use of CUDA intrinsic functions (Section 4.3.6.2) to calculate the FFT twiddle factors[3].

Dimoudi et al. timed their cuFFT implementation and compared it to `Accelsearch`. In all cases only the real-to-complex FFT, correlation, and power spectrum calculations were timed. As previously noted, the FDAS implementation of Dimoudi et al. uses interbinning whereas `Accelsearch` uses fine binning. The two methods differ computationally, complicating direct comparison of results, thus tests were performed using single bin ($r_n = 1$) as well as with interbinning. For the former, the two processes should be fairly similar, while in the latter the two will have the same frequency resolution; however, `Accelsearch` uses a more computationally intensive process to obtain it.

`Accelsearch` was run using 20 OpenMP (Open Multi-Processing) threads on the 10 CPU cores (20 Hyperthreading cores) of an Intel Xeon E5-2650. Four GPUs were used for testing; two of which, a Tesla P100 (Pascal architecture) and a Tesla M40 (Maxwell architecture), were included to give an indication of performance on top-end, compute-specific single GPU cards. In addition, two dual-GPU cards were tested: a Tesla K80 (Kepler architecture) and Tesla M60 (Maxwell architecture). When using the dual-GPU cards, only one of the two GPUs was used in testing. The authors note that dual-GPU cards are desirable as they often have better performance per watt, despite the individual GPUs often running at slightly lower clock speeds. If, on a dual-GPU card, two independent time series are processed on each GPU, it may be expected that performance will scale with the number of GPUs. Details of these four GPUs are included in Table B.1.

[3]Twiddle factors are trigonometric coefficients that are multiplied by the data in the course of the DFT algorithm.

Dimoudi et al. found that in all implementations, run time scales roughly linearly with N_Z. Although, in `Accelsearch`, performing the initial real-to-complex FFT of a 2^{23} point signal takes $220\,\mathrm{ms}$, which is a significant proportion of the overall run time. For $N_Z = 97$ and $N_t = 2^{23}$, using their cuFFT implementation on the M40 Dimoudi et al. are able to create the full r-$\dot{r}$ plane in $84\,\mathrm{ms}$ and $95\,\mathrm{ms}$ with single bin and interbinning respectively. For a sampling rate of $64\,\mu\mathrm{s}$[4], these values mean that 6380 (for single bin) and 5600 (for interbinning) full r-$\dot{r}$ planes can be created in the duration of the observation.

It was found that the speed-up of the GPU version over the CPU version decreases as N_Z is increased; this is attributed predominantly to the time it takes to perform the initial real-to-complex FFT on the CPU. For a 2^{23} point signal, using single bin, the speed-ups quickly decrease from 30 X for small N_Z (16) to 3 X for large N_Z (257). When using interbinning, the speed-ups almost double for large N_Z, with the speed-ups scaling from 30 X to 5 X over the same range. If the real-to-complex FFT is excluded from the timing and only the combined run time of the correlation and power spectrum operations are compared, the speed-ups drop to a maximum of 14 X and 17 X for small N_Z and to a maximum of 2 X and 3.5 X for large N_Z. In each of the above pairs of speed-ups, the lower value refers to single bin while the higher value refers to interbinning. The fact that the speed-up for large N_Z almost doubles is attributed to the efficiency difference between interbinning and fine binning, rather than the implementation itself. Interbinning is an approximation to a 2-bin Fourier interpolation which is applied after the correlations. This allows the segment-specific Fourier transforms used when interbinning to be half the width of those used in the fine binning equivalent, consequently almost halving the computation, and thus the run time.

The results reported above all used the cuFFT variant. Dimoudi et al. found that their custom FFT clearly benefits from the Maxwell architecture, where it runs 1.5 to 3.2 times faster than their cuFFT variant. This speed-up varies with the number of templates and filter size (ω). Their custom FFT version performs optimally for filter sizes of 1024 and 512, with wider filters performing better when larger numbers of templates are used. These improvements are not as marked in the older Kepler architecture. The better performance of the Maxwell architecture is attributed to increased shared memory capacity and bank size. Dimoudi et al. found that their custom FFT code is sensitive to shared memory efficiency, while the cuFFT code is almost entirely dependent

[4]This is consistent with the planned SKA pulsars searches.

on global memory bandwidth.

The final tests were performed on a Tesla P100 – a modern, compute-specific GPU. Without any specific tuning, they found an approximate 75% reduction in run time on this high spec GPU. The improvement of the custom FFT variant over the cuFFT variant is similar to those of the Maxwell architecture, with an improvement of 1.5 X for large numbers of templates when interbinning. To examine the real-time processing capabilities of this card, consider an SKA-equivalent observation containing 2^{23} samples with $\Delta t = 64\,\mu s$ and $N_Z = 96$. In the duration of such an SKA-equivalent observation, they are able to process 27826 and 21744 independent time series with single bin and interbinning respectively. These values scale inversely with N_Z and drop to 8600 and 6954 for $N_Z = 256$. These values are in line with the 1500 beams, and 500 to 6000 DM trials, proposed for the upcoming SKA pulsar surveys [66]. This code currently only implements the real-to-complex FFT, correlation, and power spectrum calculations. However, these results indicate that with GPU acceleration, it should be possible to run the full FDAS for three beams in real time.

In summary, Dimoudi et al. have begun implementing a GPU-accelerated version of the FDAS, intended to be used with the SKA pulsar searches. At present, this implementation only creates a large r-$\dot{r}$ plane, it does not perform any harmonic summing, searching, or CO. They find that using interbinning can almost halve the run time of a large search, although they do note that the use of interbinning on the correlated Fourier components is not well understood and this requires further investigation. They find that combining the correlation into a single CUDA kernel, as well as using shared memory and registers to avoid reading and writing intermediate values to global memory, can significantly reduce the run time of performing the correlation. The implementation created by Dimoudi et al. demonstrates the power of GPU acceleration, as well as the benefit of a thorough understanding of the architecture combined with well-crafted optimisations.

Chapter 5

Analysing the FDAS and designing a GPU-accelerated pipeline

To reduce the run time and cost of the FDAS, we intend to port the existing serial implementation, `Accelsearch`, to a GPU implementation. We aim to produce an efficient search that can be used across a wide variety of GPU architectures and search parameters with a modular and highly configurable implementation.

We followed a top-down design approach, first profiling and decomposing the existing serial CPU implementation into its main stages and constituent components, identifying their inter-dependencies. This deconstruction allowed the key data structures and memory layouts for a GPU implementation to be defined. These were then used to draft an asynchronous parallel pipeline for a GPU. In this chapter, the term *pipeline* is used with reference to the FDAS and its components (Section 2.3.4.2), as opposed to the overall pulsar search pipeline discussed in Section 2.3. Our GPU implementation is modular; it was not feasible to design the components so that they could replace, piecewise, the existing CPU equivalents, since we made some necessary functional changes to the pipeline. Even in the cases where components are similar, their inputs and outputs are in different formats and memory locations. Accordingly, a fully independent pipeline was designed and the entire pipeline was implemented, profiled, benchmarked, and optimised where necessary. We refer to our GPU accelerate port of `Accelsearch` as `AccelGPU`, detail of where the software can be accesses are in Chapter A.

In this work, we deem the input to the FDAS to be a DFT of a time series representing a single DM trial. Although `Accelsearch` can perform the real-to-complex DFT for small data sets, it usually operates on a precalculated DFT. We do not include this single long real-to-complex FFT in our implementation, as this is easily computed by one of the highly optimised existing libraries such as cuFFT [100] or FFTW [41]. As the time required to complete the FDAS of a single

DM trial is low, there is no benefit to distributing this computation across multiple computational nodes. However, most standard large-scale searches cover many thousands of DM trials; which can be distributed over a cluster, with each node handling a subset of the DM trials. However, this is beyond the scope of this work, which is focused on a single GPU-enabled multi-core computational node.

This chapter describes the analysis of the FDAS and the design of our GPU-accelerated FDAS, while the following chapter provides details on the implementation, `AccelGPU`. The first of four sections in this chapter presents a profiling analysis of `Accelsearch`, specifically focusing on the computationally intensive components of the search. The second section gives details of some fundamental modifications to the `Accelsearch` approach, which we identify as areas that hindered performance. This is followed by an outline of the design of our parallel GPU implementation of the FDAS, which spans sections three and four, the two stages being sufficiently distinct in their operation as to warrant independent sections.

5.1 CPU profiling

`Accelsearch` was profiled to determine the overall workflow and the areas where parallelism could be leveraged to decrease the run time of the FDAS. A firm understanding of the structure and components of the search, as discussed in Chapter 3, are needed here, as the components are used as the base unit of the profiling analysis. The components are groupings of some low-level, usually computationally intensive, operations functioning as a unit. These groupings were made with the expectation that some of these components could be implemented as CUDA kernels. Profiling `Accelsearch` enabled a comparative analysis of the components, allowing them to be ranked in order of optimisation priority. Furthermore, this process identified a number of fundamental modifications to the existing pipeline which benefit the new implementation; these modifications are discussed in Section 5.2.

For this analysis, version 2.0 of `Accelsearch` [112] was profiled using the `Callgrind` [86] profiling tool. This tool runs the program and, at a fixed interval, queries the location of the execution pointer in the code. Thus, the output is an approximation of the relative number of clock cycles spent in each function of an application. This can be used to estimate the proportional run time of individual functions, which can in turn be mapped to the components.

Table 5.1: Percentage of instructions spent in the various stages, tasks, and components of a CPU search of a $1e7$ sample observation of Terzan 5, summing 16 harmonics with a $Z_{\max}$ of 100.

Stage	Task	Component	Standard (%)	In-Memory (%)
CG			**93,82**	**85,22**
	Initialisation		0,07	0,02
		Coefficient calculation	0,02	0,01
		Kernel FFT	0,04	0,02
	Plane creation		65,16	19,30
		Input normalisation	0,25	0,06
		Input FFT	0,60	0,12
		Multiplication	10,23	2,95
		iFFT	41,75	12,26
		Contamination removal	3,80	1,35
		Power calculation	3,14	1,11
		Copy to in-memory		0,30
	Sum and Search		28,59	65,59
		Harmonic summing	21,30	50,30
		Search	7,24	15,15
		Candidate storage	0,05	0,15
CO			**6,18**	**14,78**
	Optimisation		6,15	14,70
		Location refinement	5,61	13,36
		Metric calculation	0,54	1,34

The components that deal with the initial candidates can have a significant negative impact on performance. Thus, all testing was done using real-world data representative of worst cases scenarios, rather than simulated data with no RFI that contains only one or two pulsars, as is common. Details of the data set used appear in Section 7.1.4.

5.1.1 Profiling

Figure 5.1 and Table 5.1 show the profile of a typical search of a $10e6$ sample observation of Terzan 5, summing 16 harmonics with a $Z_{\max}$ of 100, run as both a standard and an in-memory search. As discussed in Chapter 3, the FDAS is divided into two stages, the candidate generation (CG) stage and the candidate optimisation (CO) stage, with the former having two variants: standard and in-memory. The CG stage creates many fixed-resolution sections of r-$\dot{r}$ plane and searches these for statistically significant periodic signals. The CO stage refines the r-$\dot{r}$ locations

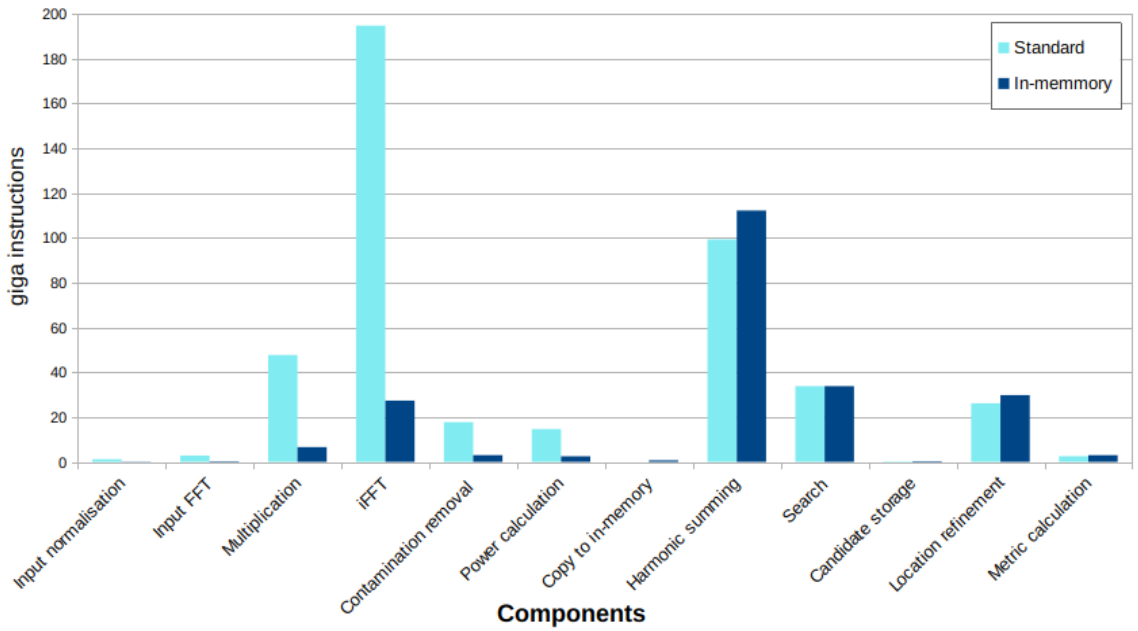


Figure 5.1: The instruction count of the components of `Accelsearch`. The search covered a 10e6 sample observation of Terzan 5, summing 16 harmonics with a $Z_{\max}$ of 100. The values shown represent the sum of a large number of instances of the component – interspersed with other components – rather than a single large block of computation.

of the candidates found in the first stage, by dynamically sampling the r-$\dot{r}$ space. The run time of the CG stage is expected to be fairly independent of the content of the data being searched, while the run time of the CO stage is directly linked to the number of, and properties of, initial candidates found and thus the content of the data being searched. Most observations are expected to produce very few candidates, however, data containing periodic RFI – such as that used here – may result in a larger number of candidates being found. Therefore, the instruction count and run time of the CO stage of this particular search may be higher than that of an average observation. Despite the high number of initial candidates found, the CG stage dominates computation, accounting for 94% to 85% of instructions. The CG stage is the best choice for initial GPU acceleration, followed by the CO stage.

The CG stage has a once-off initialisation task, the majority of which comprises convolution kernel generation. This is separated into two components: calculating filter coefficients and FFTing the coefficients. These are both computationally intensive; however, as they are performed only once, they account for only a small fraction of total computation.

The primary tasks of the CG stage are plane creation, harmonic summing, and searching. Each

section of r-$\dot{r}$ plane is created using a segment of the input DFT and a convolution kernel. These segments are normalised using median normalisation (Section 3.4.2) and then FFTed. These two components account for a very small proportion of computation, however they could still both benefit from GPU acceleration. This is followed by multiplication, which is conceptually simple. For each plane, a single complex-valued input vector is pointwise multiplied with each row of a precalculated convolution kernel. This component is ideally suited to the GPU computation model and constitutes a moderate proportion of the execution time, thus should be calculated on the GPU. Once the multiplication has been completed, each row of the plane is iFFTed. This iFFT component accounts for the largest proportion of the computation time and is ideally suited to being run on the GPU. Prior to harmonic summing, the non-contaminated complex results of the iFFT need to be converted to powers, this selection and conversion comprises $4\% \sim 8\%$ of the run time. This operation should be performed by the GPU, as the relevant values are needed on the device.

The third task has three main components: harmonically summing powers, locating local maxima that exceed some threshold, and storing the associated initial candidates. Harmonic summing and locating the maxima account for the bulk of the task and are both good candidates for GPU computation. The candidate storage component compares the output of the previous component to the collection of existing initial candidates and potentially adds these to the collection, while removing any candidates that may be "superseded" by new ones. As described in Section 3.4.4, candidates are compared using their sigma values, these are calculated in double-precision and account for a small fraction of the total computation, making the candidate storage component well suited to calculation by the CPU.

The CO stage is fundamentally simpler than the CG stage: it takes a collection of initial candidates as input and has only two computationally significant components. Prior to optimising, some initial candidates need to be *culled*. A strong periodic signal will usually result in a large "feature" in the r-$\dot{r}$ plane. Similar features may occur at multiple harmonically related positions in the r-$\dot{r}$ plane. Thus, a collection of initial candidates found by the CG stage usually contains a number of clusters of initial candidates. Only the "fundamental" candidate of each cluster should be optimised. The other initial candidates are considered "spurious" and should be discarded to reduce unnecessary computation during CO. To correctly identify the spurious candidates, the

entire collection of initial candidates is needed. This is the reason that the CO stage is dependent on the full completion of the CG stage and that candidates are not optimised during the CG stage.

Once candidates have been culled, each final candidate needs to be optimised. The dominant computational component of the CO stage is refining the location of local maxima in a harmonically summed r-$\dot{r}$ space. In `Accelsearch`, this accounts for $\sim$95% of the run time of the CO stage. In this component, many r-$\dot{r}$ points need to be calculated at arbitrary r-$\dot{r}$ locations, which requires calculating r-$\dot{r}$ values through direct convolution. By far the most time-consuming operation of these convolutions is calculating the many filter coefficients needed. This component can definitely benefit from GPU acceleration, however, the Nelder–Mead method upon which it is based is not well suited to GPU computation, meaning that an entire redesign of this component is necessary. Once the r-$\dot{r}$ location of a candidate has been refined, a number of other metrics such as the sigma value and the second derivative of the frequency are calculated. These calculations are performed once per candidate using predominantly double-precision computation, and should be performed using the existing CPU implementation.

The key computational differences between the standard and in-memory variants of the search are in the components associated with the initialisation and plane creation tasks, and are a result of the duplicated computation required to calculate the sub-planes in the standard variant (Figure 5.1). For the components of the plane creation task, the ratio of observed instruction counts between the standard and in-memory searches are within 1.4% of the expected values given in Section 7.3.1.3. The requirements for harmonic summing and searching are the same in the two variants, resulting in the search component varying little between the two. However, the same is not true for the harmonic summing component. This is a result of the additional logic required to index the harmonically related points in the full in-memory r-$\dot{r}$ plane, as opposed to the sub-planes of the standard variant. For the CO stage, there are minor differences in the amount of computation between the two variants of the search, as they do not find exactly the same candidates. This is due to the normalisation varying slightly between the two; the length of the sections of DFT used to generate the sub-planes in the standard variant are shorter than the length of the single section used in the in-memory variant. Thus, the two variants may normalise similar bins using slightly different median values. This can result in the variants finding similar candidates at slightly different r-$\dot{r}$ locations.

From the profiling it was clear that the CG stage was the best choice for initial GPU acceleration, followed by the CO stage. The standard search is dominated by the plane creation task and, in particular, the iFFT component, while the in-memory search is dominated by the sum-and-search task, specifically harmonic summing. The components were ranked in optimisation priority as follows: iFFT, harmonic summing, multiplication, search, and location refinement; the remaining components were assigned a similar optimisation priority.

5.2 Modifications to Accelsearch

In our analysis of `Accelsearch`, a number of fundamental modifications were identified which may increase the overall speed of the search. The most important of these are: using a dynamic segment size, splitting the in-memory r-$\dot{r}$ plane, and changing the precision at which the powers are stored, each of which is discussed below.

5.2.1 Dynamic segment size

In `Accelsearch`, the value of S_w is hardcoded, which can result in a large amount of wasted computation, as described below. With a set value of S_w, widening a plane to have a power-of-two width requires padding with valid r-$\dot{r}$ values that are calculated and not used. In the case of a hardcoded S_w, some values of the run-time parameter $Z_{\max}$ will result in $S_w{+}2t$ being fractionally larger than the closest power of two. In this case, ω will have to be almost double $S_w{+}2t$, resulting in a large amount of padding and therefore unnecessary computation. In `Accelsearch`, for a given $Z_{\max}$, this padding can only be reduced by altering S_w, which requires recompiling the binary. As shown in Section 7.4.1, ω can be an important factor in computational efficiency, and thus search speed. Therefore, in `AccelGPU`, both ω and $Z_{\max}$ are run-time configurable parameters, and S_w is dynamically scaled at run-time so as to minimise padding. This simple change gives better run times across a range of search parameters, and in some cases, will almost halve the amount of computation performed in the CG stage.

5.2.2 Splitting the in-memory plane

The amount of memory used is an important concern for memory-limited GPUs. The second modification involves splitting the full in-memory r-$\dot{r}$ plane and processing it as two independent halves, drastically reducing the memory requirements of the in-memory search. This is important, as the amount of memory used to store the full r-$\dot{r}$ plane is the factor which determines whether or not the faster in-memory search can be used. The r-$\dot{r}$ plane is centred on $Z = 0$ and extends from +$Z_{\max}$ to -$Z_{\max}$. Due to the multiplicative nature of harmonic summing, the full r-$\dot{r}$ plane can be split horizontally into two halves, one containing positive Z values and the other containing negative Z values. The two halves of the full r-$\dot{r}$ plane can be created and searched independently, as all harmonically related $\dot{r}$ values will always have the same sign. If the two sections are processed sequentially, this halves the memory requirements with little effect on the amount of computation. This doubles the size of the input DFT that can be searched using the more efficient in-memory variant.

5.2.3 Numerical Precision

The last modification concerns the precision at which the r-$\dot{r}$ powers are stored, which also influences the memory requirements of the search. In `Accelsearch`, the majority of computationally intensive tasks are done at single-precision, double-precision is only employed for the calculation of the coefficients used to generate the convolution kernels[1], and for the location refinement in the CO stage. Thus, the CG stage is considered to operate at single-precision. In `AccelGPU`, a similar approach was taken using predominantly single-precision with only minimal double-precision calculations where necessary. The component-specific details of the precision used is given in Section 5.3.3 and Section 6.2.3.

The precision at which the post-convolution powers are stored is a significant consideration. The harmonic summing component sums up to 16 floating-point powers from the r-$\dot{r}$ plane(s). This sum is then compared to a threshold (Section 3.4.4) to determine whether or not to consider the point as an initial candidate. These summing and comparison processes do not require particularly high precision, thus the powers can be calculated at single-precision but stored on the device at

[1] The coefficients are calculated in double-precision but Fourier transformed and stored in single-precision.

half-precision. This will result in some error in the summed powers; the maximum value of this error is the product of h and the half-precision epsilon value. Thus, the loss in precision can easily be compensated for by decreasing the threshold by the applicable value. The half-precision epsilon value is $\sim$9.77e-4 which is significantly smaller than the threshold values, thus the amount by which the threshold is decreased is usually in the range of a hundredth of the original threshold. For a standard search, this only drops the detection sigma value by less than 0.01. This small decrease does not significantly increase the number of additional candidates found.

The option to store the powers at half-precision is the third modification made in `AccelGPU`. This halves the size of the memory required to store the r-$\dot{r}$ plane(s); the reduction in size has two benefits. Firstly, it impacts the speed of memory-bound CUDA kernels, which will be shown to make up the majority of the CG stage. Secondly, it halves the memory required to store the in-memory r-$\dot{r}$ plane(s). If the powers are stored at half-precision, in combination with using a split r-$\dot{r}$ plane, this quadruples the size of the search that can be performed using the faster in-memory variant. This is a significant improvement when considering the relatively limited amount of memory available on most GPUs.

5.3 Design of GPU candidate generation

The CG stage has multiple levels at which parallelism can be exploited. At the highest level, the CG stage is fundamentally iterative for both the standard and in-memory variants, so independent iterations can be processed in parallel. In `Accelsearch`, the iterations of the standard variant are independent; however, the iterations of the in-memory variant are not. This is due to the harmonic summing in each iteration being reliant on the completion of all previous iterations to build up the full r-$\dot{r}$ plane. To overcome this in `AccelGPU`, we divide the in-memory CG stage into two sub-stages: one to create the full r-$\dot{r}$ plane, and another to search it. These two sub-stages are both iterative with independent iterations and therefore each can be run in parallel. Figure 5.2 shows a schematic representation of our implementation of the CG stage.

The iterations of the "main loops" in the CG stage all contain a different mix of the same basic components. The structure and functionality of a number of these components differ slightly from those in `Accelsearch`; the rationale for these changes is discussed below. Many components

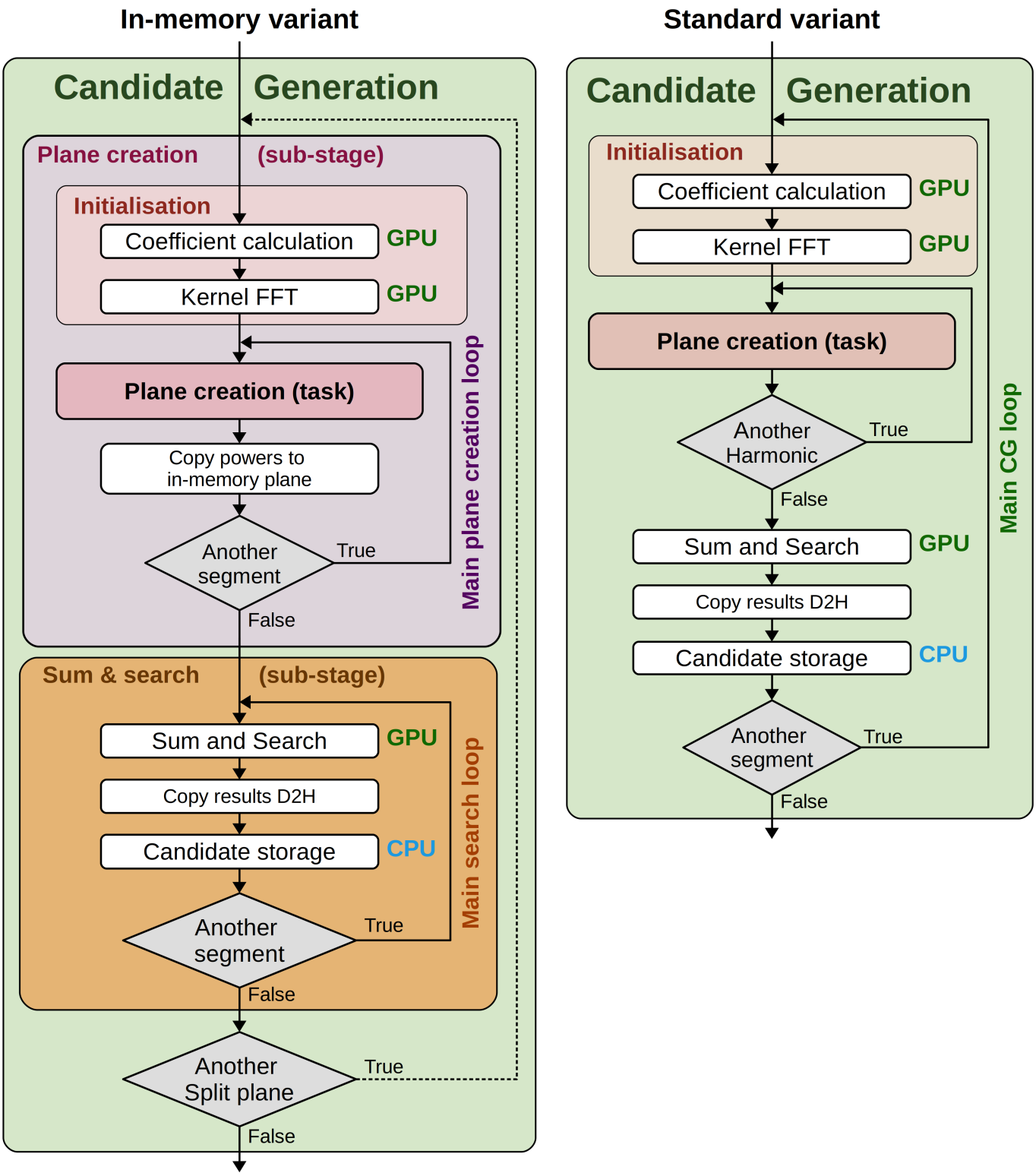


Figure 5.2: A schematic representation of the standard and in-memory variants of the CG stage. The in-memory variant is split into two sub-stages. Details of the components in the plane creation task can be found in Figure 5.3.

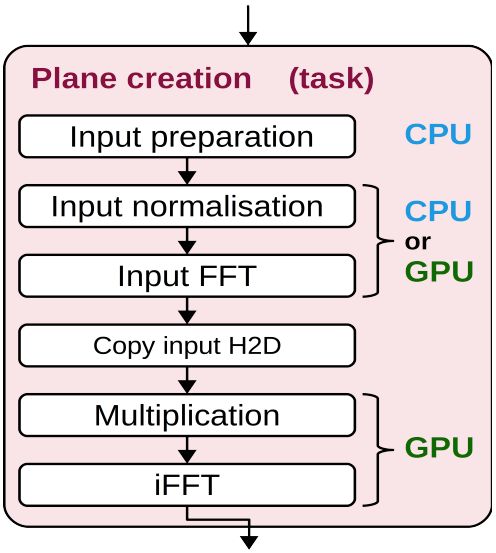


Figure 5.3: A schematic representation of the plane creation task.

have a very high degree of task parallelism and are specifically designed to be run as CUDA kernels. This will likely result in the most significant reduction in run time, as it leverages the highest degree of parallelism.

The main loops of the CG stage are iterated hundreds to thousands of times. We use a system similar to the plans of the FFTW [41] and cuFFT libraries [100] with a custom data structure referred to as a CG plan. This data structure has two elements: a description of the configuration and execution plan, and memory to hold the input, output, and all intermediate values needed to complete any one of the iterations. A single CG plan can be used multiple times, removing memory allocation and configuration from the performance-critical loops.

As discussed in Section 4.4, it is often desirable to aggregate small units of similar work into a group to be processed together, enabling latency hiding, and increasing device occupancy. The amount of work to be processed from a single segment is limited by the size of the r-$\dot{r}$ planes, which is dictated by ω and $Z_{\max}$. The value of $Z_{\max}$ is set by the structure of the specific search, and increasing ω may not decrease computation time, due to the nonlinear computational complexity of the Fourier transforms involved. Thus, the logical next option is to aggregate multiple segments together and operate on them as a single unit. This approach has the added advantage that a number of the operations of the various components can be performed on multiple segments of data simultaneously; which can amortise computation overhead. We use the term *batch* to refer to a collection of contiguous segments – and thus families of r-$\dot{r}$ planes – that will be processed

together as one unit. Thus, our CG plan data structure allows the simultaneous processing of N_s segments, grouped in a batch.

5.3.1 Parallelism

The parallel decomposition of the CG stage was performed hierarchically. At the highest level, the tasks done by the main loops of the CG stage were reordered so that the iterations would be independent. This enables coarse grain, thread level parallelism, with the concurrent processing of CG plans. Below this is asynchronous GPU execution. Running a single CG plan entails some CPU computation, GPU computation, and memory transfers to and from the device. Thus, processing multiple batches can be run as an asynchronous pipeline, allowing concurrent utilisation of the CPU, GPU and memory buses. At the lowest level, some components are run as individual CUDA kernels, exploiting the fine grain parallelism of the GPU architecture. Each of these is discussed in more detail below.

5.3.1.1 Thread level parallelism

At the highest level of parallelism, multiple batches and thus CG plans can be run concurrently. Concurrent processing of multiple CG plans on a single GPU can increase device occupancy and thus increase the speed at which a single GPU can process all batches. To allow multiple CG plans to run concurrently using a single GPU, the execution of each CG plan is processed in a separate CPU thread. It is not feasible to spawn a separate CPU thread for each batch, as there are simply too many. Rather, a small number of CPU threads (N_g) are spawned, each of which is assigned a separate preallocated CG plan. Each thread uses its plan to iteratively process a subset of the batches. To avoid the costly overhead of regular CUDA context switching, and to allow the sharing of plan-independent device memory – such as the full r-$\dot{r}$ plane – a single CUDA context is used for each device in the search.

5.3.1.2 Asynchronous GPU execution

At the intermediate level of parallelism, a number of the components required to execute a single CG plan can be run in parallel. One CPU thread iterates over a subset of the batches, each of which is processed using a single CG plan. Running a single CG plan entails a mix of: preparing

some input data with the CPU, copying this to the device, processing it by launching one or more CUDA kernels, copying some results back from the device, and processing and storing these using the CPU. This process can be regarded as a pipeline comprising CPU computation, GPU computation, and memory transfers to and from the device. Therefore, multiple batches can be processed as an asynchronous pipeline and thus, multiple components run concurrently, each operating on the data from a separate batch. An asynchronous GPU pipeline enables the simultaneous use of the two processing units and the two memory buses, allowing up to four-way concurrency in a single CPU thread using a single CG plan. To enable this concurrency, all GPU kernels and memory copies are launched asynchronously and use CUDA streams and events to manually synchronise execution. These streams and events are part of the configuration parameters of a CG plan.

This asynchronous GPU execution creates the opportunity for the CPU to concurrently perform some of the components. The main input components – normalising and FFTing the segments of the input DFT – are both good candidates for CPU computation. These components both act on the same small data set and represent a relatively small amount of computation. We implement CPU and GPU versions of both of these components. Allowing these to be performed by either the CPU or the GPU gives us a means to change the distribution of work between the two processing units. This enables `AccelGPU` to balance CPU and GPU workloads to decrease asynchronous run time.

The components in the plane creation task and the sum-and-search task are very well suited to GPU computation and operate on larger data sets; accordingly, these are implemented only on the GPU. The candidate storage component has only one computationally intensive task – sigma calculation – and requires a high degree of branching, and is thus well suited to CPU computation. The configuration of the search, notably the plane width and the $Z_{\max}$ covered, will change the relative computation time of the various components. Thus, the choice regarding which components are run on the CPU or GPU should be made at run-time with these factors taken into consideration.

Synchronous execution

It is often beneficial to be able to run a pipeline synchronously. In asynchronous execution, both concurrent CPU threads as well as the asynchronous execution of CUDA kernels within a single CPU thread, add a significant level of complexity to the application. For the purposes of validation, accurate component timing, profiling, and debugging, it is desirable to be able to run the search in a synchronous manner, so that each CG plan and all the components thereof run synchronously. In this work, such synchronous execution is allowed by means of run-time configuration.

5.3.2 CG plan memory layout

Many of the components of the CG stage are memory bound. Thus, the layout of the working memory of a CG plan is of great import, as many of the design considerations of the individual components are heavily reliant on the layout of the memory they access. Accordingly, we discuss the memory layout before detailing the individual components.

This section describes a hierarchical memory layout, which we use to store harmonically related data. This layout is relatively adaptable and is used for storing the 2D convolution kernels as well as for the complex and power values. These device memories will be used as the inputs and outputs of our CUDA kernels, and make the largest part of the working memory of our CG plan data structure. This memory layout (Figure 5.4) is hierarchical, with points being grouped into rows, rows grouped into planes, planes grouped into stacks, stacks grouped into families, and families grouped into batches, each of which is discussed below.

In a family of r-$\dot{r}$ planes, the primary plane is the largest and has the largest r value (Section 3.4.3). Each plane is required to have a minimum uncontaminated width that is the relevant harmonic fraction of S_w. However, the full width of each sub-plane is required to be a power of two, therefore some of the planes of a family will always contain some padding. This is the case even with our new dynamic segment size (Section 5.2), as this only removes padding from the primary plane. The values in these 2D planes are stored in a contiguous block of memory, in row-major order.

One of the main optimisation techniques in GPU computation is to aggregate many small units

of similar work into a group which can be processed as one larger unit. We group all the planes of the same width into a *stack*; this allows a number of the components, most notably the iFFTs, to be performed as a batched operation (Section 4.2.3). This grouping allows many of our CUDA kernels to operate on a stack rather than a plane or a single row. Figure 5.4 shows how a family of 16 planes is grouped into 4 stacks, with decreasing power-of-two widths. The green sections are the parts of each plane that are used, with the contamination shown as the blue end-portions of the planes, and the padding as the light grey regions. We align all the planes of a stack so that the used part of each one starts in the same column; this starting position is determined by the contamination in the largest plane in the stack.

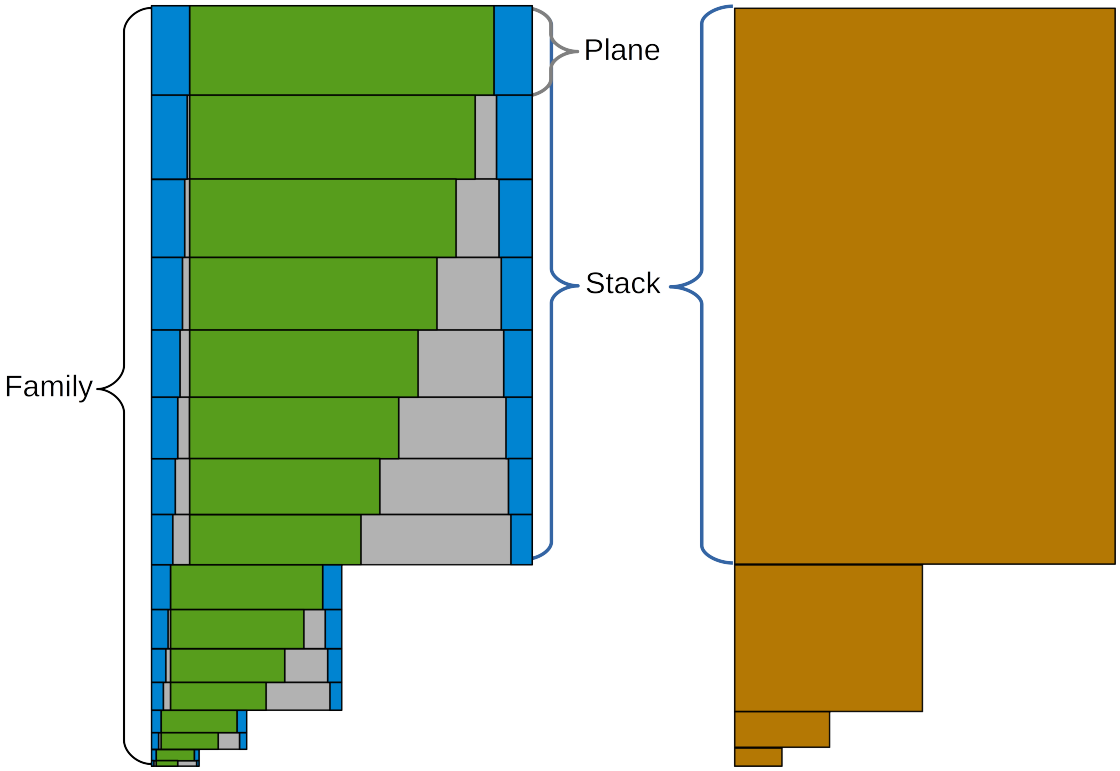
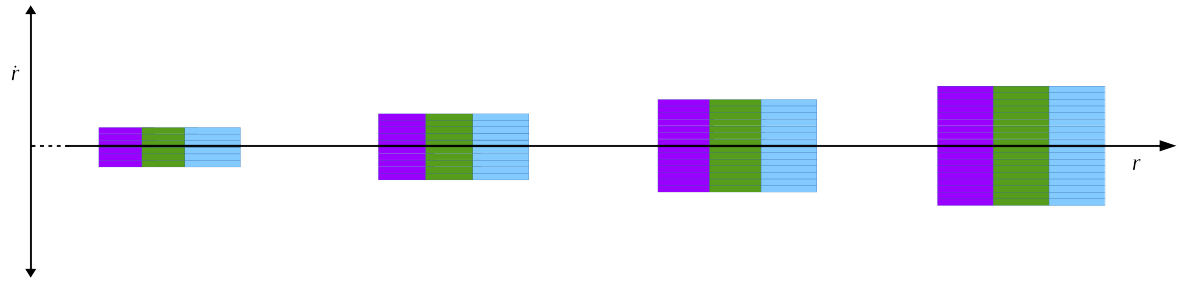
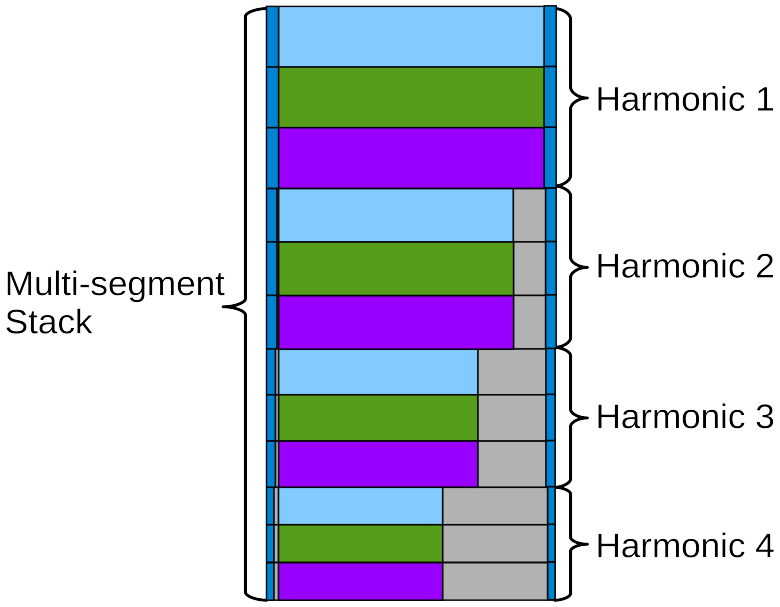


Figure 5.4: The layout of a family of 16 r-$\dot{r}$ planes in 4 stacks. The sections of the planes used is shown by the green areas; the largest plane is the primary and the others are sub-planes. The light grey areas represent the padding added to the planes, so as to have full widths that are powers of two. These padded planes are grouped together to form stacks as shown by the orange areas on the right.

Even when aggregating the planes of a family into stacks, this may not represent enough work for a CUDA kernel to saturate a GPU, especially when $Z_{\max}$ is small. To further increase the size of a stack, we use batches; these allow us to aggregate all the similar-width planes from multiple families into a single stack. The planes from the separate families that correspond to the same



(a) The layout of the planes of a batch in r-$\dot{r}$ space



(b) A stack with planes from multiple segments

Figure 5.5: The layout of a stack from a batch containing three families of planes. Each family is shown in a separate colour. Figure 5.5a shows how the segments are adjacent in r-$\dot{r}$ space, thus the planes from similar harmonics but different families are contiguous and even overlap, due to the contamination and padding of planes. Figure 5.5b shows the partitioning of a stack containing planes from four segments, the corresponding planes from the families are grouped into harmonics. The contamination is shown in blue and the padding in grey.

harmonic will all have the same width, height and padding. A collection of these similar planes is referred to as a harmonic of a stack; this grouping is shown in Figure 5.5b. In a batch, the number of harmonics in a stack (h_s) decreases as the stacks become narrower (Figure 5.4).

The layout described above is used to store the working memory of a CG plan. This memory is allocated as a large continuous block, and the layout of the stacks and planes within this, is described by the plan's configuration parameters. Two of the key parameters of a CG plan are the *plane width* (ω), which refers to the width of the widest stack in the batch, and the number of segments aggregated together to form a batch (N_s). In `AccelGPU`, ω and N_s are command-line configuration parameters, which allow the users some control over the granularity of the CUDA kernels used in the CG stage.

In summary, individual points are aggregated into rows, these are collected together in row-major order to form a plane. Multiple planes with similar widths are grouped into stacks, stacks are grouped into families, and multiple families are grouped into batches– with stacks of similar width from each family being grouped together. Thus, a batch is a contiguous block of memory containing a hierarchical collection of points from multiple families. A CG plan data structure contains multiple batches to store the input, output, complex planes and power planes; each of these is used to store all the points for all the harmonics of N_s segments.

5.3.3 Components

This section gives the functional detail and design considerations of the CG components. Figure 5.2 and Figure 5.3 show the components of the CG stage, indicating which components are performed by the CPU and/or GPU, and where the various memory transfers between the host and the device could take place. Our GPU implementation of the CG stage is very similar to `Accelsearch`. However, the tasks performed by a number of the components have been slightly altered to better suit the GPU architecture. In this section, we discuss computationally intensive components identified previously, as well as a number of the minor administrative components and memory operations needed to perform these components on the GPU. The function, input, output, and dependencies of each component are outlined below, with detail given of which processing unit will be used to perform the components.

5.3.3.1 CG initialisation

The CG stage has an initialisation task which establishes the configuration of the stage and instantiates all of the CG plans. The main initialisation tasks are: establishing the devices to be used, examining the parameters of the search and determining the execution path, determining the configuration of each component, and generating the convolution kernels. Allocating and deallocating device memory has the penalty of explicit synchronisation of the device, and is thus not to be performed in a performance-critical loop. Therefore, all the device memory is allocated, as well as the majority of the host memory that will be used in the CG stage. In addition to establishing the execution plan, a number of other initialisation tasks are performed at this point, including reading the input DFT into host memory and generating the convolution kernels.

The convolution kernels are used by the multiplication component and thus need to be stored in device memory. In `Accelsearch`, a separate convolution kernel is created for each plane in the family (Section 3.4.1). However, the convolution kernels of all the harmonics of a stack are a subset of the convolution kernel of the largest harmonic in the stack. Thus, in `AccelGPU`, only one convolution kernel – the size of the largest harmonic of the stack – is created per stack. Furthermore, only one set of these convolution kernels is stored in device memory and used by all CG plans. This amortises many of the memory reads of the multiplication component. The two computational components associated with generating these convolution kernels are discussed below.

Coefficient calculation

Creating a convolution kernel requires the calculation of many acceleration coefficients. Even though the relative cost of this once-off computation is low, the values are needed on the device, therefore this component is performed using the GPU. Since the convolution kernels are calculated only once and reused, numerical accuracy can be increased by calculating the acceleration coefficients at double-precision. However, the CG stage operates at single-precision, so they are stored in single-precision.

Kernel FFT

Once the acceleration coefficients have been calculated, every row of each convolution kernel needs to be FFTed. All the rows of the convolution kernel for a stack have the same width and can thus be grouped together and transformed in a batch operation. The cuFFT library [100] is used to perform the FFTs as an in-place, single-precision batch transformation.

5.3.3.2 Input preparation

We introduce a new component, *input preparation*, which is run by the host. The input of a CG plan is a DFT. A separate small section of this DFT is needed to generate every r-$\dot{r}$ plane used by the plan. A CG plan has a portion of memory – referred to as *input memory* – to store these sections. This is allocated as a paired set of pinned host memory and device memory, since the sections need to be transferred from the host to the device. The input preparation component copies the relevant sections of the DFT to the host input memory. When the input normalisation component is performed by the CPU, the input preparation component zeroes the host input memory prior to the memory copies.

5.3.3.3 Input normalisation

Prior to plane creation, each section of input data is normalised using median normalisation (Section 3.4.2). We implement this component on both the CPU and the GPU. In both cases, it is performed on the relevant input memory as an in-place operation.

On the CPU, the median is efficiently calculated using the Quickselect algorithm which has an average complexity of $\mathcal{O}(n)$ [108]. Efficient median calculation on the GPU can be done by sorting the values in parallel and selecting the median [15]. Each section of the DFT will have a power-of-two length, which is well suited to being sorted on a GPU using a sorting network. This normalisation can be performed as a batched kernel, operating on all the sections of input of a stack, potentially increasing efficiency.

5.3.3.4 Input FFT

After normalisation, each input section is FFTed, this can be done by the CPU or the GPU. If the normalisation is done on the GPU, the FFT should be done here as well, so as to avoid unnecessary transfer of data to and from the device. The FFT is implemented as an in-place batched operation on the relevant input memory. The FFTW library [41] is used to perform FFTs on the CPU, and the cuFFT library [100] for the GPU.

5.3.3.5 Copy H2D

Both the normalisation and FFT can be done by the CPU or the GPU. Therefore, there are three points at which the host-to-device (H2D) memory transfer can be performed: prior to normalisation, between normalisation and FFT, or post-FFT. The normalisation and FFT are both in-place operations, thus the size and location of the memory transfer is not influenced by the configuration of the computational components.

5.3.3.6 Multiplication

This component performs a pointwise multiplication of the sections of input and the convolution kernel, and is run on the GPU. It reads from the input and kernel memory, and writes to a memory location referred to here as the *complex plane*, since the results are complex values. Complex multiplication consists of only six arithmetic operations and relies on six values to be read or written from memory, therefore this component can be considered a memory-bound problem.

We considered two approaches to performing the multiplication. The first is to use a stand-alone CUDA kernel; the second is to use cuFFT callbacks (Section 4.2.3). In the latter case, the multiplication of each element can be done as a pre-transform operation to the iFFT. This approach eliminates one memory read and one write per element of the complex plane. This reduced memory access could significantly improve the performance of a memory-bound operation. We tested this method, and found it to be slower than using a stand-alone CUDA kernel, presumably due to synchronisation inefficiency in the cuFFT implementation. We therefore use a stand-alone kernel to perform the multiplication.

5.3.3.7 iFFT

The iFFTs can be performed as a batched cuFFT iFFT operating on an entire stack. This is not performed as an in-place operation, rather, data is read from the complex plane and written to the *powers plane*. Using two memory spaces increases the independence of the components, allowing a higher degree of asynchronicity in the CG pipeline.

5.3.3.8 Power calculation

We combine contamination removal and power calculation into a single operation, as this reduces memory accesses. Due to the low arithmetic intensity of power calculation – three operations per r-$\dot{r}$ value – it is considered a memory-bound problem. We calculated the powers using a post-transform cuFFT callback. The primary advantage of this is the associated reduction in memory transactions, which is the primary consideration in a memory-bound operation. Having the iFFT store powers rather than complex values halves the memory written by the iFFT kernel. Storing the powers as half-precision values compounds this, quartering the memory written. This has the knock-on effect of reducing the amount of memory read by the succeeding CUDA kernel. These reductions in memory operations make this an ideal method. However, using a cuFFT callback adds some overhead and increases the run time of the iFFT kernel. Having a post-transform callback that simply writes the complex values to the applicable memory – which is equivalent to having no callback – can increase the run time of a batched iFFT kernel by up to 35%. Thus, the overhead cost of using a callback has to be balanced against the gains from reducing memory operations. We find that the callback method is usually slightly faster than the other alternatives investigated, especially when storing the powers in half-precision and when using more modern GPUs, which can reduce the run time of the kernel by 25%. Integrating the power calculation into the iFFT makes it very difficult to differentiate, as a result, we do not show power calculation as a separate component in the analysis that follows.

5.3.3.9 Sum-and-search

We group harmonic summing and locating of the maxima into a single *sum-and-search (SAS)* component, which is performed on the GPU. Each element of a summed plane is used only once, thus, there is no need to store the harmonically summed values. In addition, candidates

with similar r values are considered to be related, and thus only the maximum power of each column of a harmonically summed plane need be returned. This allows the summed powers to be calculated, immediately compared to a running maximum, and discarded. Combining the two components significantly reduces the number of memory transactions required.

The SAS component reads values from the plan-specific powers plane or the in-memory full r-$\dot{r}$ plane. Each stage of harmonic summing produces a set of per-column maxima. Each maximum can be stored as a tuple: a summed power and a plane-specific row index. These per-column tuples are the output of the SAS kernel and are stored in *results memory*.

5.3.3.10 Copy D2H

The values stored in the results memory are required to be copied to the host, thus, this memory is allocated as a paired set of pinned host memory and device memory. As with the input, the output is comparatively small, comprising only two 16-bit values per column of the primary plane for each stage of harmonic summing. Having a large proportion of computation associated with a relatively small amount of input and output data is ideally suited to GPU computation. It is noted that if the entire powers plane of a batch were to be returned to the host, the time taken to copy this volume of data to the host would usually be greater than the total run time of all the GPU components needed to generate the values in the plane. Thus, there is little point in performing any of the components that access either the complex or the powers plane, on the host.

5.3.3.11 Candidate storage

In `Accelsearch`, the initial candidates are stored in a linked list ordered by r. Any candidates within 17 bins of each other are considered duplicates and only the most significant is stored in the linked list. We found that adding candidates to the linked list can take an unnecessarily long time, particularly if many spatially similar candidates are found. Thus, we chose rather to store the initial candidates in a constant-sized array, indexed by r. This has the advantage of easy and fast access, but at the cost of using an increased amount of host memory – roughly comparable to the amount needed to store the input DFT. The candidate storage component is run on the CPU where it reads values from the host-side results memory and writes values to the collection of initial candidates in the relevant data structure, which is the final output of the CO stage.

5.4 Design of GPU candidate optimisation

Once the spurious candidates have been culled, the remaining candidates are ranked and optimised. This optimisation consists of two components – the first refines the location of the candidate in the r-$\dot{r}$ plane, and the second calculates a number of relevant metrics of each final candidate. Once optimisation is complete, the final candidates are written to disc for later investigation, usually by a human operator. The actual run time of this stage is highly dependent on the number and properties of the initial candidates found. If a large number of candidates are found, the run time of this stage could constitute a relatively large proportion of total run time, and would thus benefit from GPU acceleration. However, the existing method of performing location refinement is not well suited to GPU computation, thus a complete redesign of this component is required. Accordingly, the section below gives extensive detail on the design of this new method.

5.4.1 Location refinement

As discussed in Section 3.4.6, the existing implementation uses two rounds of the derivative-free Nelder–Mead method [85] to refine each r-$\dot{r}$ location. This method uses a simplex which iteratively "walks" across the r-$\dot{r}$ space, converging on the relevant maximum. It has the advantage that comparatively few r-$\dot{r}$ points have to be calculated. However, it is susceptible to getting stuck at a suboptimal local maximum, or iterating for a long period of time if no clear maximum is found. The Nelder–Mead method does not have high task parallelism and has a high degree of branching, meaning that it is not well suited to being performed on a GPU. Thus, if a GPU is to be used, some other form of derivative-free optimisation is preferable. The results of the location refinement process have intrinsic value, therefore there is scope to use an optimisation method that can leverage the greater computational power of a GPU to improve both the speed and the results of this component.

The results of the location refinement process can be improved by reducing the number of suboptimal maxima that are found, and by improving the spacial accuracy at which the maxima are refined. To find the correct local maximum to refine requires the exploration of a large region of the r-$\dot{r}$ space around the initial candidate. Once the best maximum has been identified, the fine-scale location of this maximum can be resolved. In this work, a form of pattern search

[129] is used to refine the location of the maximum. A pattern search is an iterative search in which each iteration calculates some pattern of points. The location of this pattern is iteratively moved in the search space until the maximum value of the pattern is located close to its centre. The size of the pattern is then reduced and the procedure is repeated. Each round of movement localises the maximum at a given resolution, while each size reduction increases the resolution. This procedure is continued until the maximum is resolved at the desired resolution. In the classic implementation, the pattern used generally consists of only a small number of points configured in a simple shape such as a "$+$" or an "$\times$". Calculating a very small group of points is not ideally suited to GPU computation. However, the increased power of the GPU means that the number of points in the pattern can be increased. After some testing, a regularly spaced 2D grid was selected as the pattern, as this is especially well suited to the GPU computation model.

The initial grid is fairly large, covering approximately 16×16 Fourier bins and has a resolution similar to that of the points sampled in the CG stage. This grid is centred on the location of the initial candidate, and all points of the grid are calculated. Each point in the grid represents a harmonically summed power, each harmonically related r-$\dot{r}$ value is calculated by direct evaluation of the convolution (Equation 2.9) and the powers of these r-$\dot{r}$ values summed. Once the entire grid has been calculated, its maximum value is located. If the maximum is not sufficiently close to the centre of the grid, the size and resolution of the grid are kept the same, the grid is moved to centre on the new maximum and a new iteration is started. However, if the maximum is sufficiently close to the centre of the grid, the size of the grid is decreased and centred on the new maximum and a new iteration is started. Each iteration that decreases the size of the grid increases the resolution at which the r-$\dot{r}$ space is sampled. This iterative process continues until the desired level of accuracy is achieved.

This method has the advantage that the grid will "follow" features in r-$\dot{r}$ space at a specific resolution, and only refine the location once the maximum of the feature has been adequately resolved. This method comes at the cost of much greater computation, as it calculates several orders of magnitude more r-$\dot{r}$ points than the Nelder–Mead method. However, increasing the size and resolution of the r-$\dot{r}$ space searched will improve the results, and calculating a grid of points is well suited to the GPU computation model. Thus, the GPU location refinement component requires a CUDA kernel which can calculate an arbitrary resolution grid of harmonically related r-$\dot{r}$ points.

Such a kernel has a wide range of uses, thus it was decided to write a fairly generic implementation which is exposed through an API. The implementation of this grid creation CUDA kernel is discussed in detail in Section 6.1.2.

With this type of hierarchical template search, the initial iterations need not evaluate the r-$\dot{r}$ values at particularly high accuracy, as their only purpose is to resolve large-scale structures in the r-$\dot{r}$ space. Two mechanisms are used to control the accuracy of the r-$\dot{r}$ values. The first mechanism is the floating-point precision at which the calculations are done. The larger initial grids can be calculated at single-precision, only swapping over to double-precision for the final iterations. The second mechanism is the convolution accuracy (Section 3.3.2) – the width of the filter used to calculate each r-$\dot{r}$ point. The initial grids can be computed using standard-length filters, while only the final grids need be created using longer, high-accuracy filters.

If the location refinement is done using only the GPU, the CPU may be largely idle during the CO stage. Thus, the CPU can be used to do some of the location refinement computation, reducing the pressure on the GPU and thus minimising overall run time. This can be achieved by terminating the GPU template search before the desired resolution is reached. The final high-accuracy, fine-resolution refinement can then be done by the CPU using the Nelder–Mead optimisation. This CPU refinement can be initiated with a very small simplex and set to terminate after a relatively low number of iterations. This final CPU refinement is optional, is always performed at double-precision using high-accuracy acceleration filters, and can run concurrently with other CPU and GPU location refinements.

Using this mixed CPU and GPU location refinement has a number of advantages. The first advantage is that the gridded search allows a much wider range of the r-$\dot{r}$ space to be sampled, meaning that there is far less chance of settling on a suboptimal local maximum. This drastically increases the amount of computation required. However, this increased computation can be very efficiently performed on the GPU. The initial location refinement can be done at lower accuracy using single-precision computation, which is well suited to a GPU. The high-accuracy calculations can be done on the CPU, which is better suited to double-precision calculation. Thus, the CPU and GPU can be used concurrently, with each performing tasks that match their strengths. It is desirable that the location optimisation is balanced such that the CPU and GPU components have similar run times, so that each is utilised to its full extent.

In `AccelGPU`, the location refinement component is highly configurable. Location refinement may be done using the purely CPU-based Nelder–Mead method, or our new GPU pattern search with an optional, fine-scale CPU Nelder–Mead refinement. The size, resolution, precision, and accuracy of the grids in each iteration, as well as the number of iterations of the pattern search, are all run-time configurable.

5.4.2 Parallelism

The CO stage operates on a collection of independent candidates. This independence means that the candidates can be optimised concurrently. In a similar manner to the CG stage, this is done by having only a few CPU threads (N_o). Each of these iterate through a subset of the candidates, this allows a level of concurrency on both the CPU and the GPU.

A single CPU thread iterates through a collection of initial candidates, with each iteration performing two main components: location refinement and metric calculation. As in the case of the main loops of the CG stage, a single CO plan data structure can be used to optimise many candidates. In the CG stage, four-way concurrency can be achieved in each of the CPU threads. However, due to the lower number of components and the execution pipeline, this is not possible in the threads of the CO stage. GPU location refinement is an iterative process, and thus introduces a second level of iteration. Each of these sub-iterations requires some data to be prepared by the CPU, copied to the device, a grid of points calculated by a CUDA kernel, and the results copied back to the host to be processed by the CPU. Each of these sub-iterations is dependent on the location of the maximum found by the previous iteration. In addition, each task in the sub-iteration is dependent on the completion of the previous one, this prohibits concurrency in these sub-iterations. However, when optimising multiple candidates, the pipeline can still be run asynchronously; once a candidate has been localised, the candidate metrics can be calculated while the location of the next candidate is refined, allowing at least two-way concurrency. This higher level of parallelism allows utilisation of both the CPU and the GPU, and creates the opportunity to increase device occupancy.

In this way, both the CG and the CO pipelines exhibit multiple levels of parallelism, making GPU acceleration an appropriate method for decreasing the run time. This chapter shows how the principles outlined in the previous chapter can be applied to accelerate the FDAS.

Chapter 6

GPU implementation of the FDAS

This chapter outlines the details of our implementation of the GPU-accelerated FDAS. It is divided into three sections. The chapter begins with an analysis of the computational speed and numerical accuracy of a number of the low-level GPU functions used in the calculation of the various filter coefficients which the FDAS is based upon. This first section concludes with an analysis of a newly developed function to dynamically compute harmonically related sections of r-$\dot{r}$ plane. The second section outlines the implementation of the CG stage, discussing and profiling a number of the CUDA kernels that can be used to perform the GPU components. In addition, this section gives the technical details of how the higher levels of parallelism are implemented. The final section deals with the implementation of the CO stage, the main focus of this is the configuration and validation of the newly introduced r-$\dot{r}$ location refinement component. The location refinement component, and thus the performance of the CO stage, is heavily reliant on the r-$\dot{r}$ plane generation functions discussed in the first section of this chapter.

6.1 Convolution implementation

The core of the FDAS is the convolution of DFT components with some acceleration filter, as given by Equation 2.9. The calculation of these filter coefficients is the cornerstone computation upon which the entire search is based. In this section, the implementation of the convolution is discussed, focusing on the numerical accuracy and computational speed of calculating the filter coefficients and performing the convolutions on a GPU. The calculation of filter coefficients represents an insignificant proportion of the run time of the CG stage, as coefficients are calculated once and reused. However, the speed and accuracy of calculating these coefficients is of vital importance in the CO stage.

An analysis of the numerical accuracy of the filter coefficients used in `Accelsearch` has not previously been published. Thus, much of the work done in this section focuses on the numerical

accuracy of the various computations. This section introduces a new, low-level operation, which can improve the accuracy of the computation of Fresnel integrals and thus, in turn, improve that of the acceleration coefficients. In addition, we present a new acceleration coefficient estimate, which can improve the accuracy of filter coefficients for very low accelerations.

This section is structured in a somewhat bottom-up manner, as an understanding of the implementation and accuracy of the constituent computations is required to discuss the finer technical details of the implementation of the high-level convolution. Thus, a number of the base GPU functions required to calculate the individual coefficients are detailed first, followed by a discussion of the GPU computation of the various coefficients used in the search. Finally, the implementation of the actual convolution is discussed by outlining our dynamic r-$\dot{r}$ plane generation function (Section 6.1.2).

6.1.1 Coefficient calculations

This section discusses a number of the fundamental GPU functions that were developed. These are not stand-alone CUDA kernels, but rather the low-level device functions that may be called from within a CUDA kernel. These functions are generally computationally heavy and form part of many of the compute-bound kernels to be discussed later. As is so often the case, the two main concerns with our GPU implementation are speed and accuracy, as well as any possible trade-off between the two. One of the simplest tools to balance accuracy and speed is the floating-point precision of a calculation. The majority of our GPU functions are templated on data type so that the key computations can be evaluated in either single- or double-precision. In this work, the primary aim of our single-precision implementations is performance. As such, wherever possible, use is made of CUDA intrinsic functions (Section 4.3.6.2) in our single-precision calculations. These functions sacrifice some accuracy for computation speed. Conversely, our double-precision implementations are often biased towards accuracy and thus use more accurate alternatives where available.

All the functions discussed in this section relate to calculating the coefficients used in the FDAS, thus an overview is given before delving into the details. The acceleration coefficients are the complex values that are multiplied with the actual DFT bin values in Equation 2.9. Each coefficient is specific to a bin offset (Δr) and acceleration ($\dot{r}$), thus these are the two key inputs

to the functions discussed below. The acceleration coefficients are given in terms of Fresnel integrals, thus one of the first GPU functions discussed is the calculation of these integrals. The acceleration term forms part of a denominator in Equation 2.9. Thus, as will be shown below, the accuracy of the coefficients quickly deteriorates as acceleration approaches zero. In the case of no acceleration ($\dot{r} = 0$), Fourier interpolation is used to calculate a Fourier response at any arbitrary Fourier frequency [115]. This calculation can be considered to be a convolution, and is structurally similar to the acceleration convolution. Thus, when acceleration is close to zero, the convolution can be calculated using Fourier interpolation. The discussion of the individual functions begins with an examination of this similar, but simpler, calculation upon which some of the later results will be built.

6.1.1.1 Metrics

Before the GPU functions are discussed, this section gives some background on the metrics used to quantitatively measure the speed and accuracy of the functions. The functions discussed here are all considered on a purely computational basis. As such, none of the analysis in this section is concerned with the latency associated with memory operations. The memory-bound operations, which make up the majority of the critical components in the CG stage, are discussed in the next section.

Speed

The main speed metric used in this chapter is clock cycles per operation (CCpO). This metric gives the speed of only the computational component of a calculation, normalised to a single thread and calculation. This allows one to examine the computation in isolation, a key tool when comparing and optimising compute-bound problems such as those discussed in this section. Formally, CCpO is the mean number of GPU clock cycles required for a theoretical single FP32 CUDA core to perform a specific calculation. This calculation could be a single instruction such as a floating-point multiplication, a standard transcendental function such as `sin`, or a more complex function such as one of the coefficient calculations discussed below. One of the advantages of this metric is that it is independent of the hardware architecture, allowing direct comparison of specific calculations across different GPUs as well as different calculations on a single GPU. Using this metric, the theoretical run time of performing N_{O} computationally bound calculations

in parallel on a specific GPU can be obtained using $CCpO\ N_{\mathrm{O}}/N_{\mathrm{cores}}/G_{\mathrm{freq}}$ – where N_{cores} is the number of FP32 CUDA cores and G_{freq} is the clock speed (in Hz) of the GPU in question.

The speeds reported in this section are actual measurements rather than theoretical speeds, and are only applicable when averaged over a very large number of CUDA cores and operations ($N_{\mathrm{O}} \geq 8\ N_{\mathrm{cores}}$). These speeds were obtained by timing custom CUDA kernels, in which each thread called the respective operation or function a set number of times in an unrolled loop. Where a calculation required inputs, relevant values were used and these were stored in registers, so as to remove any dependence on memory access. Each kernel invocation was created with a grid large enough that there would be multiple thread blocks per SM, ensuring high occupancy. The total number of clock cycles is calculated as the product of the run time, clock frequency, and the number of FP32 cores on the GPU. The total number of operations is obtained by the product of the number of CUDA threads in the grid and the number of iterations each thread performed. Using these, CCpO is calculated as total clock cycles over total operations. It is noted that many of the calculations timed may not actually be performed by an FP32 core, some may be performed by an FP64 core or a SFU. Even in these cases, the metric has the advantage that the values generated are comparable and are relatively GPU-independent. Naturally, there may be a slight variation between different generations of GPUs, but within generations these speeds should remain fairly constant. It should be noted that this is the most optimistic use case, with no dependencies or memory operations. Therefore, this metric gives an indication of the best case computation-only speeds. However, in well-crafted, compute-bound calculations (where there is enough computation to saturate a GPU and hide the latency of the requisite memory transactions) this speed will almost entirely determine the speed of the calculation. Table C.1 gives an outline of the speeds of a number of the key base instructions, functions and intrinsic operations in CUDA.

Accuracy

The relevant accuracies were calculated by comparing values calculated on the GPU to reference values calculated on the CPU. The reference values were calculated using quad-precision floating-point calculations, often computed to a very high precision, using the Newton-Raphson method. Many of the values calculated by the various functions are continuous, thus the error values reported are maximums generated from extensive, but not exhaustive, tests.

6.1.1.2 Fourier interpolation coefficients

We implemented a device function (`fourierCoef`) which calculates an individual Fourier interpolation coefficient. Fourier interpolation (Section 2.3.3.2) is a technique that allows the calculation of a Fourier component at a non-integer Fourier frequency. This is achieved by correlating a segment if the input DFT with a matched FIR filter of length m (Equation 2.4). The coefficients of this filter (Equation 2.5) are the value to be calculated by the Fourier interpolation coefficient function. The complex components of the Fourier interpolation coefficients are:

$$\begin{aligned} \Re(I_{\Delta r}) &= \frac{\cos(\pi\Delta r)\sin(\pi\Delta r)}{\pi\Delta r} \\ \Im(I_{\Delta r}) &= -\frac{\sin^2(\pi\Delta r)}{\pi\Delta r} \end{aligned} \tag{6.1}$$

It should be noted that both of these terms are periodic and have a decreasing amplitude of $1/(\pi\Delta r)$. The `fourierCoef` function calculates coefficients in both single- and double-precision. This is the simplest of the functions discussed here and is reliant on very few basic floating-point operations and functions; its accuracy is expected to be limited by the trigonometric functions. There are two options for calculating the trigonometric calculation: multiplying Δr by π and using the CUDA intrinsic function `__sinecosf`, or using the standard device function `sinecospif`. The intrinsic function makes use of the SFU, and may be expected to use four clock cycles, whereas the more accurate `sinecospif` function uses $\sim$22 CC (Table C.1).

The error metric used to examine the accuracy of the GPU implementation, is the norm of the complex error of a coefficient. This can be thought of as the euclidean distance in the complex plane between the reference and calculated values. This metric has the advantage that it is a single positive value, and it will always be greater than both of the constituent error values, giving a reliable maximum error for either of the complex components of the value.

The speed and accuracy of the `fourierCoef` function using the two trigonometric options in both single- and double-precision are shown in Figure 6.1. The dashed line in the figure is the amplitude of the periodic terms and can be used as a proxy for the maximum size of the norm of the complex components, and thus the maximum error. The error, when using the single-

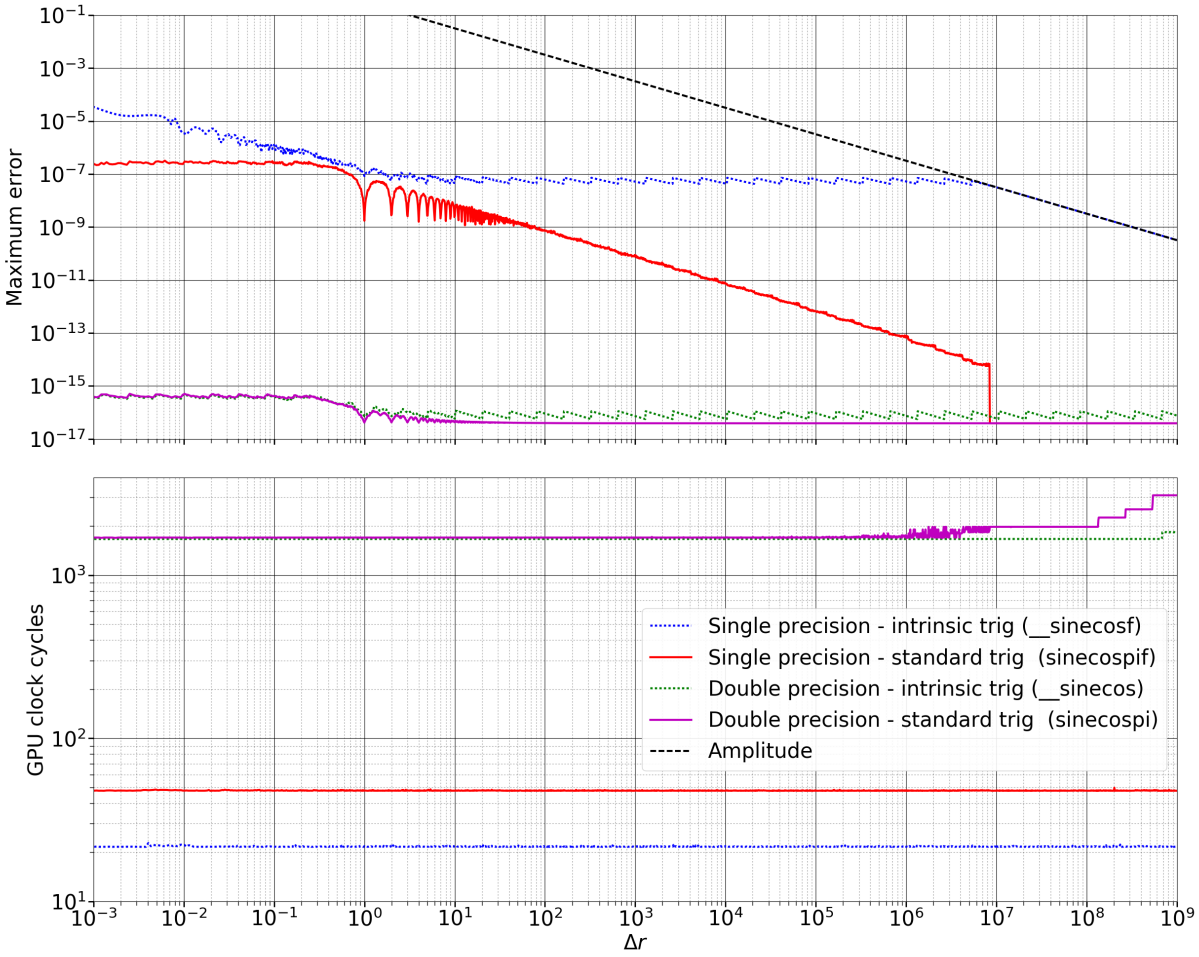


Figure 6.1: The accuracy (top panel) and speed (bottom panel) of Fourier interpolation coefficient calculation. The error values shown are the maximum error of 1e6 equidistantly spaced trials in logarithmically increasing windows. This computation can be performed in single- or double-precision, using standard or intrinsic trigonometric functions. The single-precision intrinsic variant is very fast (20 CC), however, the error relative to the amplitude increases as the filter width increases. The single-precision standard variant runs in ∼50 CC, however the relative error remains constant at approximately 6 significant figures. In double-precision, both variants have similar run times and accuracy.

precision intrinsic `__sinecosf` function, flattens off at $\sim$6e-8, with the "saw teeth" occurring at powers of two, while the `sinecospif` variant starts at $\sim$2.5e-7 and then drops off fairly linearly for values greater than one. The pronounced scalloping between one and a hundred marks the shared roots of the two components, at which both functions tend to zero, reducing the error. The rate of decrease in the error is the same as the amplitude, showing that the relative error is roughly constant at just over six significant figures, close to the precision limit of the single-precision representation. The drop in the error just below $\Delta r = 1\mathrm{e}7$ marks the value $\Delta r = 2^{23}$, which is the point beyond which a single-precision value will always be represented as a whole number. Beyond this point, a single-precision representation of Δr has insufficient precision to function correctly. However, at this point, the magnitude of the coefficient will always be less than 1e-7, which may be considered insignificant. In most use cases, the width of the FIR filter, m, is expected to be significantly less than 2^{23}. The error in the variant using the intrinsic function is fairly constant, thus showing a increase in relative error, to the point at which the error matches the amplitude at $\Delta r \simeq 8\mathrm{e}6$. However, in standard uses, m is expected not to exceed $5\mathrm{e}2$, at which point there are still roughly four significant figures. The error in both double-precision variants are approximately constant, with `sinecospi` being only slightly more accurate.

Both single-precision versions of this function can be seen to have a constant run time of $\sim$22 and $\sim$48 CC. These values may appear somewhat high, as the implementation consists of approximately seven operations, excluding the trigonometric function; this may be due to some of the overhead required to time these functions. The difference of twenty-six clock cycles is in line with the timing of the `__sinecosf` and the `sinecospif` functions (Table C.1). This disparity, though doubling the clock cycles, is still fairly minor. The run times of the double-precision variants are significantly higher at $\sim$1670 CC, showing the expected performance bias towards single-precision operations. This speed difference is in line with the 1:32 ratio of FP32 to FP64 cores in the GeForce range of GPUs used in these tests.

For both single- and double-precision variants of the `fourierCoef` function, the slower but more accurate standard trigonometric variant is used. In single-precision, run times are low and the relative error in the intrinsic variant is significant for larger values; in double-precision, higher accuracy is obtained with comparable run times.

6.1.1.3 Fresnel integrals

An acceleration coefficient is given in terms of the pair of normalised Fresnel integrals $\mathcal{C}(x)$ and $\mathcal{S}(x)$ (Equation 2.10). In `Accelsearch`, evaluation of Fresnel integrals is done with a code released in the Cephes Math Library [123]. This implementation was adapted to run efficiently on a GPU, taking into consideration a number of factors affecting speed and accuracy. In the implementation, the evaluation is always done on the absolute value of the input, as the Fresnel integrals are antisymmetric and thus the signs of the results are simply inverted where needed. The evaluation is divided into three cases: one for small inputs ($x \leq \sqrt{2.5625}$), another for large inputs, and the last for the asymptotic behaviour for very large inputs. These three cases are implemented as the device functions $\texttt{Fres}_1$, $\texttt{Fres}_2$ and $\texttt{Fres}_3$. Each implementation simultaneously calculates a pair of Fresnel integrals ($C(x)$ and $S(x)$). In the case of small inputs ($\texttt{Fres}_1$), the integrals are approximated by a power series, calculated using two sixth- and two seventh-order polynomials. This calculation can be done with approximately 55 basic arithmetic operations and 25 constant coefficients.

$\texttt{Fres}_2$ covers the intermediate range of input values and accounts for the vast majority of Fresnel integrals used in the acceleration coefficients. Thus, much attention was focused on the efficiency of this function. $\texttt{Fres}_2$ uses Equation 6.2 which makes use of two auxiliary functions, $f(x)$ and $g(x)$. These auxiliary functions are a power series, calculated with four polynomials: one each of tenth, eleventh, twelfth and thirteenth order. The evaluation of these utilises approximately 115 basic arithmetic operations as well as a paired trigonometric evaluation. Accordingly, this method may be expected to be at least twice as computationally intensive as $\texttt{Fres}_1$. The third case, $\texttt{Fres}_3$, simply returns 0.5, the asymptotic value around which $\mathcal{C}(x)$ and $\mathcal{S}(x)$ oscillate. This function is largely ignored as it is practically never used in `AccelGPU`.

$$\begin{aligned} \mathcal{C}(x) &\approx 0.5 + f(x)\ sin\left(\frac{\pi x^2}{2}\right) - g(x)\ cos\left(\frac{\pi x^2}{2}\right) \\ \mathcal{S}(x) &\approx 0.5 - f(x)\ sin\left(\frac{\pi x^2}{2}\right) - g(x)\ cos\left(\frac{\pi x^2}{2}\right) \\ &\text{for: } x > \sqrt{2.5625} \end{aligned} \tag{6.2}$$

Our GPU implementations of $\texttt{Fres}_1$, $\texttt{Fres}_2$ and $\texttt{Fres}_3$ are templated on data type so that the Fresnel integrals can be evaluated in both single- and double-precision. The maximum error and computation speed for a number of variants are shown in Figure 6.2. Similar to the previous section, the error metric used is the norm of the error in the two paired Fresnel values. The blue and green lines show the results when using only standard single- and double-precision operations. The transitions between $\texttt{Fres}_1$ and $\texttt{Fres}_2$ can clearly be seen at $\sim$1.6. For $\texttt{Fres}_1$ and $\texttt{Fres}_2$, in both single- and double-precision, there is a steady increase in error with the size of the input. The error in the double-precision values increases roughly linearly from 3e-19 to 5e-9 for an input in the range 1e-3 to 1e8. In the case of the single-precision implementation, the errors in $\texttt{Fres}_1$ and $\texttt{Fres}_2$ reach a maximum of approximately 4e-7 and 2e-4 respectively.

The output of $\texttt{Fres}_1$ is calculated using four polynomials. Thus, as expected, the run time remains constant at about 48 CC. Note that the number of clock cycles (~ 49) is, in fact, below the number of arithmetic operations (55); this is due to the large number of fused multiply-add (FMA) operations that can be used in calculating a polynomial, each of which counts for only a single clock cycle and two arithmetic operations. $\texttt{Fres}_2$ uses four polynomials as well as a trigonometric operation. It can be seen that for the single-precision implementation, only 80 CC are required for the calculation. Again, the clock cycles are less than the 115 arithmetic operations due to the many FMA operations used.

As in the previous section, the amplitude of the periodic Fresnel values is shown as the dashed line. This amplitude is the largest value of the norm of the two Fresnel values; thus, as before, the error should be considered relative to this value. It can be seen that once the various errors intersect this amplitude, they decrease with it. These intersections mark the points at which the error "saturates" the results. From these points, the amplitudes of the periodic functions are correct, however, all accuracy in phases is lost. Therefore beyond these bounds, $\texttt{Fres}_3$ can be used.

The single-precision calculation of $\texttt{Fres}_2$ is very fast at only 80 CC; however, the relative error is large and increases with the size of the input to the point at which the error saturates the results at an input size of only $\sim$4100. The coefficient calculations can have Fresnel inputs well beyond this point, thus the error in the standard single-precision implementation is considered too high for use in this work. This error is predominantly caused by the single-precision trigonometric

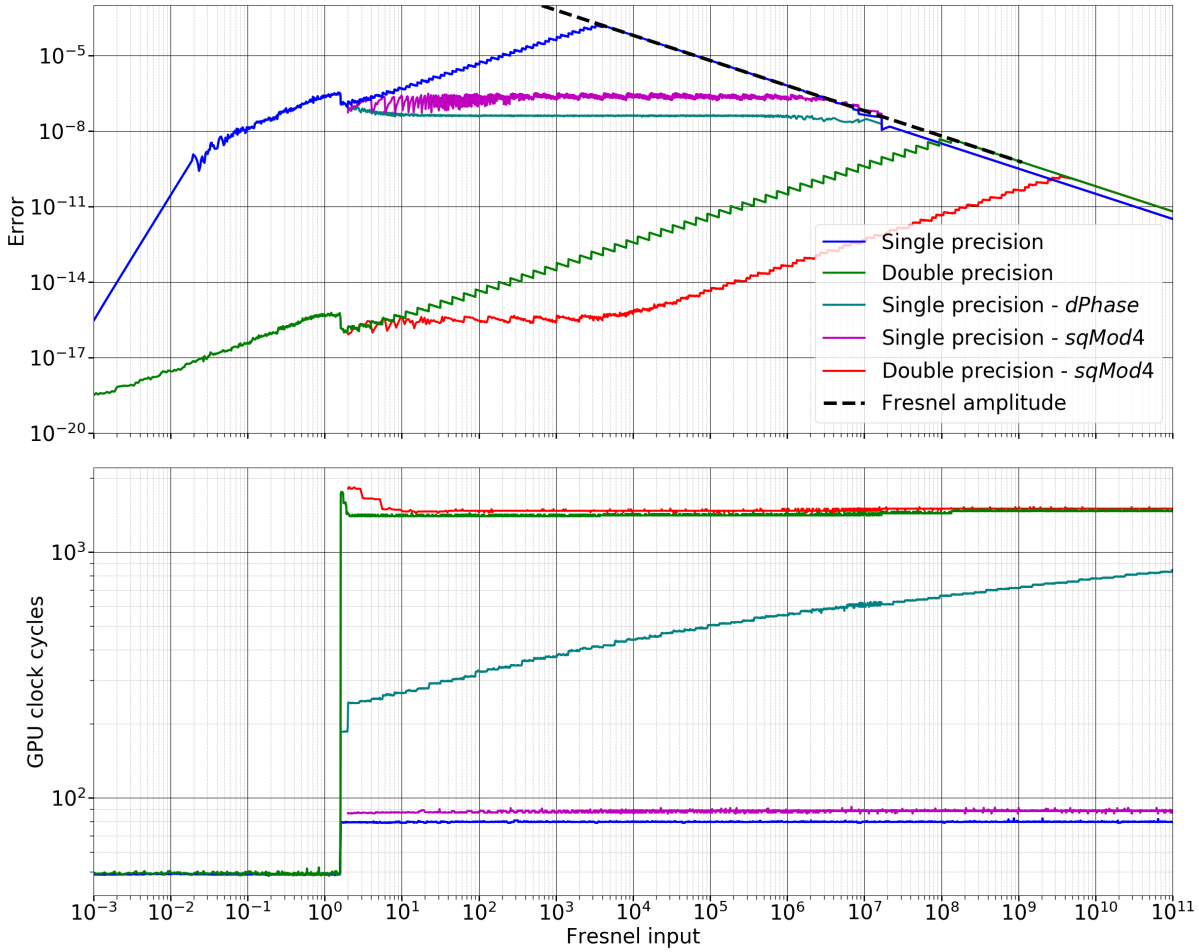


Figure 6.2: The speed (top panel) and accuracy (bottom panel) of calculating Fresnel integrals. The single-precision variant runs very fast, however, the error increases rapidly as the input increases. Fixing this error using double-precision phase calculation decreases the relative error, however, it increases the computation time by more than 200 CC. Using the `sqMod4` function to perform the phase calculation similarly decreases the error, while maintaining a very low run time (90 CC). The double precision variant using the `sqMod4` function increases the accuracy while maintaining a similar speed.

evaluations of very large values. $\texttt{Fres}_2$ has a paired trigonometric operation on $\pi x^2/2$. This term increases quadratically with x causing the large error values. One approach to reclaiming some accuracy is to perform some of the critical calculations in double-precision, specifically the square and trigonometric operations. However, double-precision trigonometric operations are particularly computationally intensive (Table C.1); the double-precision trigonometric `sinecos` can easily take 680 CC, which is about nine times more than the entire single-precision implementation of $\texttt{Fres}_2$. An alternative is to accurately calculate the "phase" of the term and use this in a single-precision trigonometric function. The trigonometric value has a period of $4x^2$, thus the phase of the term can be calculated as $x^2 \pmod 4$. Therefore, the value of $x^2 \pmod 4$ can be calculated in double-precision, the result converted to single-precision and used in a single-precision trigonometric function. This method requires only a single double-precision multiplication and modulus operation (`fmod`), and is shown as the teal line in Figure 6.2. The double-precision phase calculation effectively drops the error to an approximately constant value of 4e-8 – a significant reduction of more than four orders of magnitude for very large values of x, and pushes the saturation point to $\sim$1.7e7. However, the double-precision floating-point modulus operation is still very costly and, as a result, the computation time can be seen to increase drastically: from approximately 80 CC to 250–600 CC.

The sqMod4 function

We developed a novel technique which calculates the phase at high precision using only eight basic 32-bit instructions. This method relies on the fact that the period of the function is a power-of-two multiple of π. This allows one to exploit the binary representation of the value to be squared. Any value (x) can be broken down into two parts (a and b) such that $x = a + b$ and a is the largest integer multiple of two less than x. Thus, $x^2 = a^2 + 2ab + b^2$ and a^2 will be an integer multiple of four, meaning it contains no phase information and can be discarded from the operand. Note that $2ab + b^2$ may well be greater than four, so is not only the remainder, but contains the entire remainder and thus all the phase information. The a^2 term usually counts for the majority of the size of x^2, thus $2ab + b^2$ is relatively small and therefore maintains a high precision representation of the phase. The beauty of our method is that the terms a and b can be obtained very efficiently, directly from the IEEE 754 floating-point representation of x. The floating-point exponent p can be used to obtain a by simply zeroing the last $24 - p$ bits of

the mantissa. It is then possible to use a to calculate b from x. A hand-tuned parallel thread execution (PTX) function (`sqMod4`) was written, which calculates the value $2ab + b^2$ for both single- and double-precision floating-point numbers. These are included in Appendix D. The single-precision variant of this function consists of only eight basic 32-bit instructions and runs in approximately nine clock cycles (Table C.1). The result of this function can be used in the single-precision trigonometric function of $\texttt{Fres}_2$. This variant is shown as the magenta line in Figure 6.2. It should be noted that `sqMod4` will only function as expected for $x > 2$, thus these results start at 2. It can be seen that the error, when using the single-precision variant of `sqMod4`, oscillates but is relatively flat and remains below 4e-7. Beyond $x = 4$, the maximum error when using `sqMod4` is always below that of the straight single-precision implementation. Thus, the use of this function reduces the error in $\texttt{Fres}_2$ in a fraction of the computational time required for even the most limited number of double-precision operations. Similarly, the double-precision variant using `sqMod4` can be seen to reduce error with a minimal increase in clock cycles. It is noted that in the existing CPU implementation of `Accelsearch`, the standard double-precision implementation – the green line in the figure – is used. Therefore, the use of the double-precision `sqMod4` operation can increase the numerical accuracy of the Fresnel integrals beyond that of the existing implementation by up to two orders of magnitude.

As with the Fourier interpolation of the previous section, the paired trigonometric function in $\texttt{Fres}_2$ can be performed using the standard or intrinsic device functions. Both were investigated, and it was found that in single-precision, the standard function decreased the error by $\sim$1e-7. In this case, it was deemed that this decrease did not warrant the additional 25 CC. Thus the $\texttt{Fres}_2$ function uses intrinsic trigonometric functions, as well as `sqMod4` for values of x greater than eight in both single- and double-precision, keeping the absolute error below 4e-7.

In summary, we developed a very efficient GPU function to calculate Fresnel integrals in single-precision. This calculation takes approximately 90 CC to compute and will have an absolute error below 4e-7, for input values below 36974. The double-precision implementation takes in the order of 4000 CC to compute and has an increasing error with a maximum of 3e-12 for input values below 36974.

6.1.1.4 Acceleration coefficients

This section includes an analysis of the accuracy and efficiency of the device function used to calculate acceleration coefficients. These values are the complex terms multiplied with the $\mathcal{F}_k$ in Equation 2.9. The coefficients are a function of $\dot{r}$ and Δr, and are given in Equation 6.3, which is obtained from Equation 2.9.

$$r(\Delta r, \dot{r}) = \frac{1}{\sqrt{2\dot{r}}} \left([S\cos(t) + C\sin(t)] + i\,[S\sin(t) - C\cos(t)] \right) \tag{6.3}$$

As before, Δr is the distance, measured in Fourier bins, of some integer DFT bin to the potentially non-integer bin location in question. The term t is the input to the paired trigonometric functions and is given as: $t = \pi/\dot{r}\ (\Delta r - \dot{r}/2)^2$, with $S = \mathcal{S}(Z) - \mathcal{S}(Y)$, $C = \mathcal{C}(Y) - \mathcal{C}(Z)$, given $Y = \sqrt{2/\dot{r}}\,[\Delta r - \dot{r}/2]$ and $Z = \sqrt{2/\dot{r}}\,[\Delta r + \dot{r}/2]$. The functions $\mathcal{C}(x)$ and $\mathcal{S}(x)$ are the normalised Fresnel integrals given in Equation 2.10.

To simplify the expression of the real and imaginary components, the values Y' and Z' are defined as: $Y' = \sqrt{2/|\dot{r}|}\,[\Delta r - \dot{r}/2]$, $Z' = \sqrt{2/|\dot{r}|}\,[\Delta r + \dot{r}/2]$ and $S' = \mathcal{S}(Z') - \mathcal{S}(Y')$, $C' = \mathcal{C}(Y') - \mathcal{C}(Z')$. From these terms, the complex components of a single acceleration coefficient for a given Δr and $\dot{r}$ are:

$$\begin{aligned} \Re(r(\Delta r, \dot{r})) &= \frac{1}{\sqrt{2|\dot{r}|}} \left(S'\cos(t) + \left(\frac{\dot{r}}{|\dot{r}|}\right) C'\sin(t) \right) \\ \Im(r(\Delta r, \dot{r})) &= \frac{1}{\sqrt{2|\dot{r}|}} \left(S'\sin(t) - \left(\frac{\dot{r}}{|\dot{r}|}\right) C'\cos(t) \right) \end{aligned} \tag{6.4}$$

From Equation 6.4 it can be seen that to calculate a single complex acceleration coefficient requires one paired trigonometric operation, a square root, the evaluation of two pairs of Fresnel integrals, and a number of basic floating-point operations. Note $\dot{r}/|\dot{r}|$ is simply the sign of $\dot{r}$. As with the Fresnel integrals, the focus of our single-precision implementation is on maximising computational speed while maintaining as low an error as possible; while the double-precision implementation focuses more on accuracy at the expense of computational speed.

The dominant computational components of these values are the two pairs of Fresnel integrals

followed by the square root, and the paired trigonometric operation. Both the accuracy and the speed of the coefficient calculations will be examined in the Δr-$\dot{r}$ plane, initially focusing on the Fresnel integrals and then extending the analysis to the acceleration coefficients.

Fresnel integrals in the Δr-$\dot{r}$ plane

Previously, the Fresnel integrals were discussed in terms of a generic input value. Here, that analysis is extended to examine the accuracy and speed of calculating Fresnel integrals, using the two inputs of the acceleration coefficient calculation. Thus, in this section, results are shown in the Δr-$\dot{r}$ plane.

When considering acceleration coefficients, the inputs of the two pairs of Fresnel integrals are: Y' and Z'. The analysis here will be limited to just one of these, Y', since Y' and Z' are symmetric about the line $\Delta r = 0$ in the Δr-$\dot{r}$ plane. Figure E.1 shows the value of Y' in the Δr-$\dot{r}$ plane; it is evident that the magnitude of Y' and Z' increase towards $\dot{r} = 0$. As a result, it is anticipated that the largest errors in the Fresnel integrals, and consequently in the acceleration coefficients, will occur at accelerations close to zero. This is of some concern as the majority of pulsars are found at low accelerations.

The bounds of the high- and standard-accuracy convolution filter lengths are drawn, as the solid and dotted red lines, in all figures of the Δr-$\dot{r}$ plane. A set of filter coefficients can be thought of as a collection of unit-spaced values on a horizontal line between the two relevant boundary lines. Thus, the area enclosed by these lines represents the range of coefficients used by `Accelsearch` and `AccelGPU`, and demarcates our region of interest (ROI) in the Δr-$\dot{r}$ plane. As such, attention is focused on the accuracy and computation speed of the values inside these bounds. Note that the filter widths at accelerations close to zero are intentionally widened, well beyond the bounds of $\dot{r} = \pm 2\Delta r$. Meaning the ROI contains some points where $\Delta r \gg \dot{r}$, causing some of the accuracy issues discussed later in this section.

As discussed in the previous section, the computation of the Fresnel integrals is split into three cases, conditional on the size of the input. The boundary points between these implementations, when expressed in terms of Δr and $\dot{r}$, describe lines in the Δr-$\dot{r}$ plane. These lines are shown in Figure E.2 and can be seen to demarcate a number of distinct regions. The dashed line marks the boundary where the transition from using $\texttt{Fres}_1$ to using $\texttt{Fres}_2$ is made. The area where

$\texttt{Fres}_1$ is used is relatively small, accounting for ∼5% of values in the ROI. The dash-dotted line marks the boundary, from which the `sqMod4` phase calculation is used in $\texttt{Fres}_2$. The obvious bands in the error values in this region are caused by the oscillation in the error of the `sqMod4` function. The value of Y' is zero at the line $\dot{r} = 2\Delta r$ and increases asymptotically towards the $\dot{r}$-axis. Thus, the error close to $\dot{r} = 2\Delta r$ is very low and increases as we move away from this line; it increases up to a maximum of ∼3e-7 at the point at which the transition from $\texttt{Fres}_1$ to $\texttt{Fres}_2$ is made. Here, the error drops slightly and rises to a similar level at the point from which `sqMod4` is used. Hereafter, the error oscillates between ∼2e-8 and ∼3e-7. This means that in the ROI, the error varies but generally remains in the relatively narrow band of 2e-8 to 3e-7. Figure E.3 is the equivalent plot for double-precision calculations. The error has a very similar structure but is approximately nine orders of magnitude smaller. The actual results of the Fresnel integrals oscillate about the asymptotic value of ± 0.5. Therefore, the errors can be judged relative to this value, showing that both the single- and double-precision Fresnel integrals are close to the precision limit of the two representations. It should be noted that this "flat" error is due to the new `sqMod4` function. If it were not used, the error in parts of the ROI would rise as high as 1e-4 and 1e-13 in the single- and double-precision calculations.

Figure E.4 shows the speed of calculating the single-precision Fresnel integrals in the Δr-$\dot{r}$ plane. The same bounds as above are demarcated, and it is evident that the speeds in the various regions are fairly constant. The single-precision $\texttt{Fres}_1$ calculation takes approximately 50 CC and the $\texttt{Fres}_2$ calculation 80 or 90 CC, depending on whether `sqMod4` is used. The majority of the ROI is covered by $\texttt{Fres}_2$, thus it can be assumed that calculating a pair of single-precision Fresnel integrals will take ∼90 CC. The double-precision clock cycles are shown in Figure E.5. These are approximately 16 times higher than single-precision. This is a substantial increase, resulting in an average of ∼3500 CC for the double-precision calculations.

Accuracy

In the previous section, the Fresnel integrals were examined in the Δr-$\dot{r}$ plane and it was demonstrated that error remains in a relatively narrow band. This section examines the accuracy of the acceleration coefficients which are calculated using these functions. Empirical testing revealed that the errors in the acceleration coefficients are not flat; rather they rise as acceleration approaches zero. The acceleration coefficients calculated directly from Equation 6.4 employing

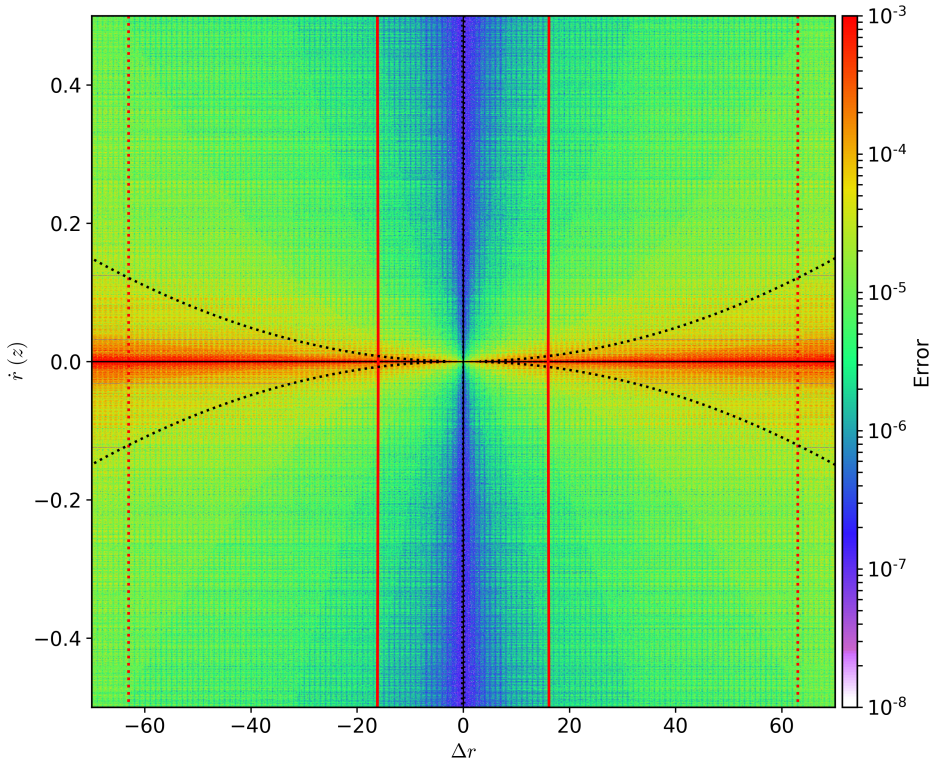


Figure 6.3: The error in the single-precision raw acceleration coefficients at low accelerations. The dotted line shows the point from which the double-precision phase calculation is used. Despite this correction, the error grows rapidly as the acceleration approaches zero.

the Fresnel integrals discussed previously, are referred to as *raw* acceleration coefficients. It is noted that, as with $\texttt{Fres}_2$, this calculation contains a trigonometric operation acting on the term, t, which in turn contains a square. Here, the value squared is $(\Delta r - \dot{r}/2)$; this is divided by $\dot{r}$, meaning that t grows very large as $\dot{r}$ approaches zero. Unfortunately, one cannot employ the same technique used in the `sqMod4` function, as in this case, the squared term is divided by $\dot{r}$, not a power of two.

This means that one cannot exploit the binary representation of the value to easily calculate the phase. However, as described in the previous section, a double-precision phase calculation can be used, in which $(\Delta r - \dot{r}/2)^2/\dot{r} \pmod 2$ is calculated in double-precision and the result converted to single-precision. This can then be used in a single-precision trigonometric function. In the ROI, t only grows to a size that warrants this, when $\dot{r} \to 0$. The raw acceleration coefficients are calculated using this double-precision phase calculation where necessary.

The reader is reminded that the filter widths are disproportionately wide for $\dot{r}$ close to zero, thus the ROI contains values well beyond the bound $\dot{r} = \pm 2\Delta r$ for small values of $\dot{r}$. In this range, even with the double-precision phase calculation, the error in the raw coefficient rises asymptotically as $\dot{r}$ approaches zero. The error in the raw coefficient for small $\dot{r}$ is shown in Figure 6.3. The dotted line in this figure shows the point at which the double-precision phase calculation is made. It can be seen that in spite of this correction, the error rises steeply as $\dot{r} \to 0$, to a point that is considered too high for the purposes of this work.

In the range where $\dot{r} \to 0$ and this large error is present, we developed new more accurate approximation for the acceleration coefficients. This approximation is a power series in $\dot{r}$, the complex components of which are given in Equation 6.5.

$$\begin{aligned}
\Re(r(\Delta r, \dot{r})) \approx & \frac{\cos(\pi\Delta r)\sin(\pi\Delta r)}{\pi\Delta r} \\
& + \dot{r}\,\frac{\sin(\pi\Delta r)}{2\pi\Delta r^2}\left(\cos(\pi\Delta r) - \operatorname{sinc}(\pi\Delta r)\right) \\
& + \dot{r}^2\,\frac{\cos(\pi\Delta r)}{4\pi\Delta r^3}\left(\frac{3}{\pi\Delta r}\left(\cos(\pi\Delta r) - \operatorname{sinc}(\pi\Delta r)\right) + \sin(\pi\Delta r)\right)
\end{aligned} \tag{6.5}$$

$$\begin{aligned}
\Im(r(\Delta r, \dot{r})) \approx & \frac{\sin^2(\pi\Delta r)}{\pi\Delta r} \\
& - \dot{r}\,\frac{\cos(\pi\Delta r)}{2\pi\Delta r^2}\left(\cos(\pi\Delta r) - \operatorname{sinc}(\pi\Delta r)\right) \\
& - \dot{r}^2\,\frac{\sin(\pi\Delta r)}{4\pi\Delta r^3}\left(\frac{3}{\pi\Delta r}\left(\cos(\pi\Delta r) - \operatorname{sinc}(\pi\Delta r)\right) + \sin(\pi\Delta r)\right)
\end{aligned}$$

The error of the raw and approximated single-precision acceleration coefficients are shown in Figures E.7 and E.8 respectively. It is clear that the error in the raw coefficients increases as $\dot{r} \to 0$ while the error in the approximated coefficients decreases; the error in the approximated coefficients grows very large in the regions where the raw coefficients have their lowest errors. The dashed line in the two figures represents the boundary $\pm\dot{r} = 0.028 + 0.0325|\Delta r|^{1.25}$ – which was found by inspection – where the mean error in the raw coefficients is approximately equivalent to the mean error in the approximated coefficients. In the ROI, the maximum single-precision error along this bound does not exceed 8e-6 and 2e-6 for the raw and approximated coefficients

respectively. Thus, if the transition between these two methods is made at this point, it may be expected that the maximum error in this part of the ROI will be in the range of 8e-6. In these figures, the dotted line close to the $\dot{r}$-axis is the double-precision phase calculation point of the raw coefficients, which is well within the region where the approximation is more accurate. This means that raw acceleration coefficients using the double-precision phase calculation need not be used if our new approximation is utilised for small $\dot{r}$. A similar bound was found for the double-precision coefficients; it is significantly smaller at $\pm\dot{r} = 0.00015 + 0.0002256|\Delta r|^{1.25}$. It is noted that at $\dot{r} = 0$, the approximation is equivalent to the Fourier interpolation coefficients given in Equation 6.1.

Our API exposes a number of CPU and GPU functions which can be used to calculate all the coefficients discussed above. The most useful of these calculates *generic* coefficients in both single- and double-precision. This function automatically selects whether to use the raw or the estimated acceleration coefficients, using the bounds discussed above. When $\dot{r}$ is sufficiently small, the function returns Fourier interpolation coefficients. Figures E.9 and E.10 show the error of these generic coefficients. The transition between the raw and estimated coefficients is clearly visible in the single-precision case, while in double-precision, this bound is only visible in the inset. It is evident that, in the case of double-precision, there is still a significant increase in error for values of $\dot{r}$ close to zero. This error increases up to a point of approximately 1e-12; thereafter, the approximation becomes more accurate. With the exception of this difference, the error in both is structurally similar, increasing across the ROI from the line $\dot{r} = 2\Delta r$ towards the line $\dot{r} = -2\Delta r$. In the vicinity of this line, they reach a maximum value of $\sim$2.4e-5 and $\sim$2e-14 in the single- and double-precision variants respectively. Outside the lines $\dot{r} = \pm 2\Delta r$, the error quickly drops off to levels of 2e-7 and 2e-16 – values close to the respective precision limits.

Figure E.6 shows the maximum error in the acceleration coefficients, which span the region between high-accuracy bounds. These values represent the maximum possible error in a coefficient for a given acceleration. The steep increase in the error of the raw coefficients as $\dot{r}$ approaches zero is clearly seen, highlighting the need for our new approximation. The error in the approximation starts flat at small values of Δr, marking the precision limits. At some point this error begins to rise steeply, intersecting the error of the raw coefficients at an $\dot{r}$ of $\sim$1e-3 and $\sim$2e-1 for single- and double-precision respectively. The generic coefficients are equivalent to the approximate and

raw coefficients for small and large values of $\dot{r}$. However, there is a region in which a filter will comprise both raw and approximated coefficients. The error in generic coefficients is relatively flat in this range. The maximum error in these sections is found close to the transition point between the two methods and is consistent with the maximum errors found along the bounds. These bounds are shown on the insets in Figures E.9 and E.10. The raw error has an upturn starting at $|\dot{r}| \simeq 10$. Below this point, the maximum error is found on, or close to, the extremes of the filter width, shown as red lines in the figures. Beyond this point, the maximum error is found close to the line $\dot{r} = \pm 2\Delta r$.

In summary, the maximum error in the generic single-precision acceleration coefficients remains in a fairly narrow band close to the precision limit and does not exceed 1e-5 for $|\dot{r}| < 200$. The maximum error in the double-precision coefficients has a slight increase for 1e-5 $\lesssim |\dot{r}| \lesssim 10$ but remains below 2e-12.

Speed

Through use of our new approximation and `sqMod4` phase calculation, it is possible to avoid using any double-precision operations in the single-precision coefficient calculations. This results in a very efficient single-precision GPU coefficient calculation which, as shown above, maintains a relatively low error, comparable to the precision limit. The speeds of the generic coefficient calculations are shown in Figures E.11 and E.12. Each coefficient requires the calculation of two pairs of Fresnel integrals: one centred on $\dot{r} = 2\Delta r$ and the other on $\dot{r} = -2\Delta r$. The boundaries relevant to the calculation of the Fresnel integrals are shown in these figures. For the majority of the ROI, a single-precision generic acceleration coefficient takes approximately $\sim$220 CC to compute. This drops to $\sim$130 CC when $\texttt{Fres}_1$ can be used for one of the pairs of Fresnel integrals. Close to the Δr-axis, the power series approximation is used, dropping the computation to only $\sim$40 CC. For future analysis, we will use the following approximation for the speed of calculating a generic coefficient: 240 clock cycles per coefficient (CCpC).

It is noted that across the full Δr-$\dot{r}$ range, the run times of each of the single-precision coefficient calculations are all less than that of performing even a single double-precision phase calculation. This highlights the benefit of the two new developments of this work, which allow the calculation of accurate coefficients using only basic 32-bit operations. The double-precision variant runs in

$\sim$3500 CC for the majority of the range, dropping to $\sim$2000 where $\texttt{Fres}_1$ can be used for one pair of the Fresnel integrals. Again, the layout of the speeds in the Δr-$\dot{r}$ plane is structurally similar, but approximately 16 times slower than the single-precision variant.

The reader is reminded that for the timings shown here, all the threads of many warps calculated the same value, and the run times were averaged over many iterations of a highly efficient loop. In most realistic use cases, each thread of a warp would calculate a separate coefficient. There is a small amount of branching, both in the calculation of the Fresnel integrals and in the choice of coefficient calculation. Therefore, real-world speeds such as those discussed in the next section, may be expected to be lower due to warp divergence.

6.1.2 Dynamic r-$\dot{r}$ plane creation

In the CG stage, fast convolution is used to create many small sections of r-$\dot{r}$ plane. Using this method, the size and resolution of each plane section are the same. These limitations, however, mean that the same convolution kernel can be used to create many plane sections, which can drastically reduce calculation time. There are situations in which sections of r-$\dot{r}$ plane with arbitrary location and resolution need to be generated, such as the planes of the location refinement component (Section 6.3.3.1) of the CO stage. Thus, a multipurpose r-$\dot{r}$ plane generation function (`rr_plane`) was developed, that can be used to efficiently calculate harmonically related r-$\dot{r}$ values on the GPU. The sections of r-$\dot{r}$ plane referred to here are defined as a 2D grid of regularly spaced points in the r-$\dot{r}$ space; each of the r-$\dot{r}$ points of harmonically related planes are the appropriate integer multiple of their corresponding r-$\dot{r}$ point in the fundamental plane. In this work, the term *size* is used to describe the scale – measured in bins – of a plane in the r-$\dot{r}$ space, whereas *dimension* is used to describe the number of points along an axis of the grid, and *resolution* refers to the spacing of the points. Thus, a harmonically related plane has the same dimension as the fundamental plane, while the size and resolution are the appropriate multiple of the fundamental. The primary use case for the `rr_plane` function is to create planes which are expected to be smaller than a single plane of the CG stage, with an envisaged size of the fundamental plane in the range of $\sim$1e-5 to 1e3 bins with dimensions in the range of 8 to 2048 points.

The `rr_plane` function is designed as a host function, meaning it is called from host code and returns values in host memory. The function takes as input the memory location of the full DFT

and the location, size, and dimension of the fundamental plane. To allow a broad range of use, the function is capable of returning three types of values: either the harmonically summed powers of the fundamental plane, separate complex values for all r-$\dot{r}$ points, or powers for all r-$\dot{r}$ points. The function begins by extracting the relevant sections of the DFT; normalising these on the host and then copying these values – as a single small strided block – to device memory. Thereafter, an appropriate CUDA kernel is launched to calculate all the r-$\dot{r}$ values, which are written to device memory. Once the kernel has completed, the results are copied back to the host for further processing.

Calculating a 2D grid of independent values is ideally suited to GPU computation. In order to calculate grids of harmonically related r-$\dot{r}$ values, we implemented two CUDA kernels. The first is referred to as `pln_nrw`, in which a separate CUDA thread is assigned to each r-$\dot{r}$ point of each harmonic plane. Each thread calculates a single r-$\dot{r}$ value, which requires calculating tens to hundreds of complex acceleration coefficients, each of which is multiplied with the appropriate input and the result added to a running sum. This kernel is the more flexible of the two, as it can create sections of r-$\dot{r}$ plane of any size and resolution, at any location. However, this requires the separate computation of each individual coefficient used, which is a computationally intensive task. This is sometimes unavoidable, however, in some cases there is scope to reuse coefficients, as with the next kernel.

The second CUDA kernel (`pln_brd`) can be used when the width of the section of r-$\dot{r}$ plane is wider than one DFT bin. As discussed in Section 6.2.3.1, each coefficient is only dependent on the value of $\dot{r}$ and Δr; the actual location that is evaluated only affects the actual DFT values multiplied with the coefficients. Thus, if two points have the same $\dot{r}$ value and their r locations are an integer number of bins apart, they will have identical coefficients which are multiplied with differing input values. As a result, when the size of the plane section is significantly wider than one bin, one can use the same coefficients to calculate multiple points in the plane. This, however, has the restriction that the points must each be separated by an integer number of bins, putting a constraint on the resolution of the section of the r-$\dot{r}$ plane created.

In `pln_brd`, a CUDA thread is assigned to each point of each plane. However, in this kernel, groups of consecutive threads are combined into *cooperatives* in such a way that all the threads of a cooperative are in the same warp. Each cooperative calculates a group of points at integer-

spaced r locations and identical $\dot{r}$ values. The convolution loop of each thread is divided into a nested loop; the outer loop iterates over sections of the convolution, with the sections being the same width as the number of threads in the cooperative. In each of these outer iterations, each thread of the cooperative calculates only a single coefficient. In the inner loop, these coefficients are then shared within the cooperative using highly efficient shuffle operations (Section 4.2.2.1). For each inner iteration, each thread gets a coefficient from the registers of a thread in the cooperative; reads relevant input value; multiplies the two, and adds the result to a thread-specific running sum. This is highly efficient, as few additional registers are required and the computationally intensive coefficient calculations can be shared amongst the threads of a cooperative. It is noted that the input values used by the inner loop of a cooperative are the same. We investigated sharing these values between threads using a similar shuffle technique. However, it was found that simply reading the input values directly from device memory was more efficient. This stems from the fact that these memory transactions are automatically cached, whereas the shuffle operations increase both register use and computational complexity, especially where the points are spaced by more than one bin.

The maximum size of a cooperative is limited to 32 threads, the size of a warp. To allow efficient decomposition of multiple smaller cooperatives into a single warp, the number of threads in a cooperative is a power of two. This dictates the number of r points and thus the width of the section of plane generated, while the number of cooperatives determines the r resolution. Thus, the width of the planes that the `pln_brd` kernel can generate is limited to powers of two ≤ 32. However, using multiple invocations of `pln_brd`, any plane with a width that is the sum of some powers of two and has a resolution of the form $(a2^b)/n \mid a, b, n \in \mathbb{N}$ can be generated. Thus, if the `rr_plane` function uses the `pln_brd` kernel, the function is allowed some leeway to slightly change the size and dimension of the plane created. This freedom allows the selection of a size and dimension that decomposes well into the constraints mentioned above. Thus, the *output* parameters of the planes generated by the `rr_plane` function do not always match the *target* parameters if the `pln_brd` kernel is used. The changes are made in such a way that the output r resolution is always the same as, or slightly higher than, the target resolution. In addition, output width of the plane will usually be increased to some appropriate width slightly larger than the target.

In the `rr_plane` function, when the target width is less than one, `pln_nrw` will be used, while if it is greater than one, `pln_brd` can be used. As will be shown, `pln_brd` is more efficient but this comes at the cost of slightly altering the output parameters of the plane. If the output parameters cannot be changed, the less efficient `pln_nrw` kernel can be used to generate planes.

Use

If a host function with GPU components, such as the `rr_plane` kernel, were to be called only once, the various resources needed – especially GPU memory – would have to be dynamically allocated and deallocated. The total run time of this process may well be longer than simply having done the calculation on the CPU. If, however, many planes are going to be created, a collection of both GPU and CPU resources need only be allocated once and reused to generate all the planes. This type of multi-plane creation is envisaged to be the primary use case of the `rr_plane` function. Thus, the main variant of the `rr_plane` function uses a data structure which encapsulates a collection of preallocated resources. This data structure is similar to the plans used in the CG stage, and is passed to the `rr_plane` function as the primary input parameter.

Speed

The aim of this section is to provide a means to estimate the time required to calculate a given section of r-$\dot{r}$ plane using the `rr_plane` function. Along the way, the performance of the two CUDA kernels mentioned previously will be characterised.

There are a number of parameters that affect the planes generated: size, dimension, h, accuracy, and the $\dot{r}$ location of the plane. To allow comparison across all combinations of these parameters, it is helpful to be able to express a plane in terms of some base unit. A natural first choice for this would be the number of individual r-$\dot{r}$ points. However, each r-$\dot{r}$ point requires a convolution, the length of which can vary drastically, and as a consequence, the computation of individual r-$\dot{r}$ points is not directly comparable. A convolution consists of repeatedly multiplying a complex coefficient with some value and summing the results. Therefore, the newly introduced convolution operation (COP) – defined as the multiplication of a value with a coefficient – was chosen as the base unit in which to express r-$\dot{r}$ planes. Any r-$\dot{r}$ plane can be expressed in terms of the number of COPs required for its computation – in this work, these cover a wide range from $1\mathrm{e}4$

to 1e13.

The full formalisation of expressing a plane in terms of COPs is not given here. However, the number of r-$\dot{r}$ points can be calculated as the product of the two dimensions and h. The number of COPs used to calculate each specific r-$\dot{r}$ point is proportional to the length of the convolution, which is a function of accuracy and $\dot{r}$ (Equation 3.2). In this analysis, planes with a square dimension were used, thus the number of r-$\dot{r}$ points is quadratically proportional to the dimension and linearly proportional to h. The number of COPs is, however, not strictly linearly related to h as the harmonic planes have larger values of $\dot{r}$. Therefore, the number of COPs increases disproportionately with h.

Using COPs as the base unit, the plane creation speed metric used is clock cycles per convolution operation (CCpCOP). This is defined as the number of GPU clock cycles needed for a single CUDA core to perform a single COP. This is a natural extension of the CCpC used in Section 6.1.1.4 and indeed, if a COP requires the calculation of a coefficient, one may expect them to be very similar. However, as was discussed previously, not all convolutions require explicit calculation of all coefficients, since in some cases common coefficients can be shared. Thus, one may expect that CCpCOPs will at times be less than CCpC and indeed, the degree to which the speed is below the mean of $\sim$220 gives an excellent measure of the efficiency of the computation.

Figure 6.4 shows the speed of the `rr_plane` function. The values reported here are meant to represent the results of the typical use case, in which many planes are created concurrently using preallocated resources. Thus, these results are obtained from running multiple concurrent iterations of the `rr_plane` function using a single GPU and averaging the results. Each iteration includes all the steps of plane generation, namely: host-side normalisation of the input, data transfers to and from the device, and GPU computation. The asynchronous nature of the execution means that the total run time is expected to be determined almost entirely by the limiting factor, which is GPU computation. Thus, the speed of the `rr_plane` function is equivalent to the speed of the CUDA kernel(s) used to calculate the r-$\dot{r}$ values.

Figure 6.4 is thus used to discuss the speed of the `rr_plane` and `pln_nrw` kernels. As expected, the speed of the `pln_nrw` kernel is very close to the average CCpC of $\sim$220, with a very slight decrease in efficiency as more harmonics are summed. It can be seen that for planes wider than one bin, the efficiency of the `pln_nrw` kernel drops slightly as the width of the input plane

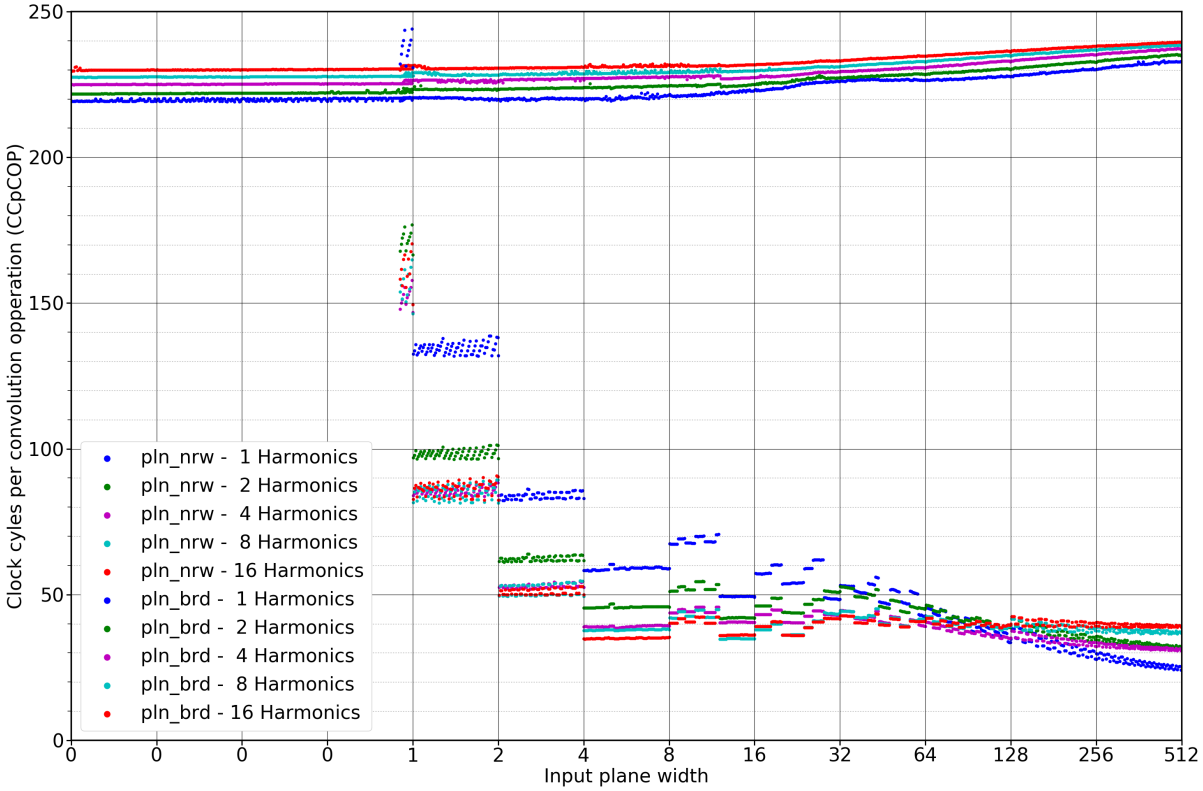


Figure 6.4: The CCpCOP of the `rr_plane` function with a target dimension of 512×512. The asynchronous run time of this function is almost entirely determined by the GPU computation, therefore this can be used as a proxy for the run time of the CUDA kernels used. The CCpCOP of the `pln_nrw` kernel is similar to the average of the generic coefficients. The `pln_brd` kernel, in which common coefficients can be shared within a warp, can be up to ten times more efficient for large plane widths.

is increased. This is a result of Figure 6.4 being generated using a constant target dimension. Thus, as the plane width is increased, the memory access pattern becomes more dispersed, as a wider range of input values is used. This reduces the cache hits, which decreases efficiency.

The main factor that influences the efficiency of `pln_brd` is the number of threads in the cooperatives, which is determined by the plane width. To see the efficiency of the `pln_brd` kernel in relative isolation, the CCpCOPs of a range of planes with power-of-two widths are examined. At these widths, the threads per cooperative match the plane width exactly. Thus the CCpCOPs are 232, 132, 83, 59, 49 and 48 for cooperatives of 1, 2, 4, 8, 16 and 32 threads respectively. It is apparent that increasing the number of cooperatives drops the CCpCOP. Increasing the cooperative size from one to two threads approximately halves the clock cycles, showing the expected increase in efficiency. The efficiency continues to increase as more threads are included, how-

ever the rate of this increase diminishes to the point at which the efficiency gain from sixteen to thirty-two threads is negligible. The speed of the `pln_brd` kernel appears to settle at between 22 and 44 CCpCOPs for wide planes, which is five to ten times more efficient than the equivalent `pln_nrw` kernel. It is important to note that the increase in efficiency between the `pln_brd` and `pln_nrw` kernels does not always translate to an equivalent reduction in run time. When using the `pln_brd` kernel, some padding may be added to the output plane; the CCpCOPs are calculated using the output dimensions. As a result, the increased efficiency has to be balanced against the time needed to compute the additional padding.

The speeds shown in Figure 6.4 are the final results taken after the `rr_plane` function automatically selected plane parameters in an attempt to balance the computational efficiency and padding, in order to give the lowest run times. These changes in parameters account for the various steps in the speeds when using the `pln_brd` kernel. It was found that for plane widths of less than eight, the efficiency gains are sufficient that the optimal width is the next largest power of two. This results in fairly constant speeds between the power-of-two widths for planes narrower than eight bins. For planes larger than eight, the best run times may be obtained when the final plane is created by combining multiple sub-planes. As an example, let us examine the planes with widths between eight and sixteen. The speeds in this range have a number of steps which can be divided into two classes: a single large step at 12, and a number of smaller steps between 8 and 12. The large step at 12 is caused by a change in output plane width. Planes narrower than 12 are generated with two sub-planes – one of width 8 and another of width 4. The combination of these two has a final speed of $\sim$68 CCpCOPs, while planes wider than 12 are created with a single plane of width 16, which is more efficient at $\sim$50 CCpCOPs. The planes narrower than 12 could be created with this more efficient kernel, however, the large amount of padding required results in a slower run time. The smaller steps in the speeds between 8 and 12 mark the points where the output dimension is changed so that the output resolution matches, or exceeds, the target resolution.

6.2 Candidate generation implementation

This section covers the technical details of our implementation of the CG stage. It begins with an outline of the layout and lifespan of the various GPU memories used, followed by a techni-

cal description of the individual components, and concludes with the details of the higher-level parallelism.

A CG plan data structure is used to complete an iteration of any one of the main loops in the CG stage. This data structure encapsulates both the host and device working memory, as well as the configuration describing how the iteration and all of its constituent components will be performed. As such, the majority of the functions which perform the various tasks and components of the CG stage take a pointer to a CG plan as their primary input.

6.2.1 Dynamic segment size

In Section 5.2 it was indicated that it is preferable to have a dynamic segment size, calculated from the user-specified ω and $Z_{\max}$. The objective of a dynamic segment size is to minimise the padding in the various planes, and thus reduce wasted computation in the CG stage. The padding for all the planes in a batch is indicated by the light blue sections in Figure 5.4. The amount of padding in the planes of a single family can be used as a proxy for the amount of excess computation needed to obtain the final powers. As discussed in Section 3.4.2, the segment size is dependent on the plane width and the half-width; the latter is, in turn, dependent on $Z_{\max}$, as given by Equation 3.2.

One may anticipate that to minimise the excess padding in a family of planes, the segment size should be determined from the primary plane as $s = w - 2h$. There is, however, a complicating factor: the relationship between $Z_{\max}$ and half-width is not linear for $Z_{\max}$ below $\sim$250 (Equation 3.2). If the relationship were linear, it would mean that the plane representing the harmonic fraction of 0.5 would be the first plane in the second stack, as is the case in Figure 5.4. However, the nonlinear relationship between half-width and $Z_{\max}$ means that this plane can sometimes be marginally too wide to be in the second stack, thus pushing it up into the widest stack. As a consequence, this plane is almost half padding, significantly increasing the total amount of padding. By decreasing the segment size a small amount, this plane will move into the second stack. Reducing the segment size will introduce some additional padding in all the planes of the first stack. However, this additional padding is less than the amount saved in the "half" plane, reducing total padding. This can be extended to the first planes of the other stacks; however, the padding in these appears to have relatively little effect on speed. The optimal segment size is thus defined

as the largest value at which the plane, representing the harmonic fraction of 0.5, is placed in the second stack.

6.2.2 GPU resources

GPUs generally have far less memory than the host they are operating on, especially when considering desktop cards. With these limited memory environments in mind, both the standard and in-memory GPU pipelines were developed to use GPU memory as sparingly as possible, while at the same time being able to scale to make full use of the available memory.

As mentioned in Section 4.2.2, it is desirable to allocate device memory in large continuous blocks that are aligned and strided. In Section 5.3.2 we describe how the planes of batch are grouped into stacks, each of which have a distinct width. To ensure all the stacks are in continuous memory, a single large block is allocated, and striding of the stacks-specific sections is manually managed.

6.2.2.1 Convolution kernels

The convolution kernels comprise one complex value for each point of an r-$\dot{r}$ plane section. Only one convolution kernel is required per stack. All the convolution kernels for a search – one for each stack – are stored in one continuous section of GPU memory in a flat, strided array of single-precision floating-point tuples. This memory is allocated and populated during the CG initialisation and is freed at the end of the CG stage.

6.2.2.2 CG plan memory

The amount of working memory used by a CG plan is of some import. One of our primary aims is to minimise this working memory. The number of plans used in the CG stage can be scaled to utilise a greater proportion of the device memory. The working memory of a CG plan encompasses a number of large blocks of device and host memory as well as a number of pointers to the various sub-regions in these blocks. As introduced in Section 5.3.3, each CG plan requires four main sections of device memory: input, complex planes, powers planes, and output. All the CG plans and associated memory are allocated once during the CG initialisation and are freed at the end of the CG stage.

The input memory is used to store a segment of the input DFT for each section of r-$\dot{r}$ plane used in the plan. The input for a single r-$\dot{r}$ plane is stored as an array of single-precision floating-point tuples the same width[1] as the r-$\dot{r}$ plane. Thus, the input memory of the CG plan is a number of arrays of varying lengths. The internal layout of individual arrays is similar to that shown in Figure 5.4, with all the similar-length arrays grouped together into a stack. The CG plans input memory is the initial device memory accessed by and iteration, therefore this needs to be populated by the host. Accordingly, this memory is allocated as a paired block of device and pinned host memory.

The complex planes are stored as single-precision floating-point tuples. These planes use the same stride as the input memory and each section has a height the same as the relevant stack. The internal layout of the individual stacks and planes is shown in Figure 5.4. The use of a continuous block allows all memory accesses for a CG plan to be relatively localised.

As discussed in Section 5.3.3.8, the data type stored in the powers plane can be one of several floating-point types: single-precision tuples or singletons; or single half-precision values. These planes are created with the same width as the complex planes. It is noted, that this is not strictly necessary, as the power planes need not incorporate the contaminated ends of the complex plane. However, having them the same size drastically simplifies indexing when using cuFFT callbacks, which – as discussed in Section 6.2.3.5 – can increase efficiency.

The output of the search components is a number of initial candidates. Each candidate is stored as a tuple consisting of a single-precision floating-point value and a 16-bit integer. The floating-point value stores a harmonically summed power, while the integer stores the row index, which expresses the acceleration of the candidate. As discussed in Section 6.2.3.6, each column of the primary plane can generate a single candidate per stage of harmonic summing. Thus, the output memory consists of an array of candidates for each stage of harmonic summing, with the width of these arrays being the width of the uncontaminated section of the primary plane. This output memory is allocated as a paired block of pinned host memory and device memory. The size of the input and output memory do not scale with $Z_{\max}$, while the complex and powers planes do. Therefore, the input and output memory are generally significantly smaller than the complex and powers planes.

[1]The width of the DFT segment is usually half that of the relevant input array, due to the zero padding required by the fine binning.

In the case of the in-memory search, the: input memory, complex plane and powers plane requirements are the same as for the standard search of a single harmonic. However, output memory differs slightly, as discussed in Section 6.2.4.2. Each CG plan requires two sections of output memory to be used in a flip-flop manner. One of these sections is allocated in the same manner as the standard search; for the other section, the memory used for the complex plane is repurposed as it is only required in the plane creation task.

6.2.2.3 In-memory r-$\dot{r}$ plane

The in-memory search requires a relatively large amount of GPU memory to store the full r-$\dot{r}$ plane. The amount of available memory is usually the limiting factor as to whether or not an in-memory search can be performed. As discussed in Section 5.3.3.8, the in-memory plane can consist of either single- or half-precision floating-point powers. The choice of type determines the size of the in-memory r-$\dot{r}$ plane, with half-precision powers requiring half the space. The memory for the full r-$\dot{r}$ plane is allocated as a large strided block of device memory during initialisation and is freed at the end of the CG stage. The r-$\dot{r}$ plane may be split into top and bottom halves. If this is the case, the memory for only one half is allocated and reused to search both halves.

6.2.2.4 Shared memory

The majority of the GPU kernels of the CG stage are memory-bound. However, each memory element is read only once, thus there is little need for shared memory. Shared memory is occasionally used; the details of this can be found in Section 6.2.3.

6.2.3 Components

6.2.3.1 Convolution kernel creation

The Convolution kernels are created asynchronously on the GPU during the CG initialisation, which allows the CPU to continue with other host-side initialisation tasks. Calculating convolution kernels comprises two separate components: coefficient calculation, and a batched FFT. Each of these require a separate CUDA kernel. These two components are closely related so both are discussed here.

To perform coefficient calculations, a CUDA thread is assigned to each point in a 2D convolution kernel. Those threads that fall in a location where a coefficient is found – shown by the red areas in Figure 3.3 – calculate a single complex coefficient using the generic coefficient calculation function (Section 6.1.1.4), and write the result to device memory. All other threads simply write a zero tuple to device memory. Once all coefficients have been calculated, the entire kernel plane is FFTed using an asynchronous batched cuFFT call. This is done during the CG initialisation by the first GPU initialised.

Similar to the existing implementation, the final convolution kernel values are stored as single-precision tuples. However, in `AccelGPU`, the precision at which the calculations are performed is run-time configurable. The run time of the coefficient calculations is comparatively negligible and the results are of great import. Thus, by default, they are calculated using double-precision. The precision of the FFT is separately configurable. It was found that the precision of this computation has less impact on the final results. Thus, by default, they are done in single-precision. If the FFTs are done using double-precision, the coefficients are calculated and stored as double-precision tuples in temporary device memory. This is due to the FT being performed as an in-place transformation which requires a separate CUDA kernel to convert the double-precision values to single-precision.

6.2.3.2 Input normalisation

The segments of the input DFT used to generate the various planes of a batch range in length from 128 to 16384 elements. The number of similar-length segments in a single batch can range from 1–72, depending on N_s and the number of planes in a stack. Normalisation of a single segment requires three tasks: calculating the powers, finding the median of these powers, and multiplying each complex element by the derived normalisation factor. Median calculation is by far the most computationally intensive of these.

On the GPU, median calculation is performed by sorting the powers and then selecting the central value. The sorting is performed by a sorting network which needs some form of communication and synchronisation, and therefore requires that each segment be handled by a single thread block. For all sizes, the entire set of powers is able to fit into the standard 48 KB of shared memory available per thread block, allowing the use of shared memory for median selection. An in-place

Bitonic Sort was used, which has constant complexity of $O\left(n \log\left(n^2\right)\right)$ [14]. This type of search iteratively sorts and merges sections of values, thus each iteration results in increasingly larger, power-of-two length sections of sorted values. The basic operations are pairwise comparisons and swapping. It is desirable to decompose the network so that each element is assigned to a single CUDA thread. However, there are at most 1024 threads per thread block [91], thus each thread can be accountable for up to eight elements. Each warp handles up to 256 (32×8) elements. The incremental nature of this process means that the sorting of sections of up to 256 elements can be done by a single warp. Sorting intra-warp elements requires no synchronisation and makes use of the butterfly shuffle operation [91], which allows the exchange of register values between threads. This significantly reduces inter-thread communication overhead.

Sorting inter-warp elements requires synchronisation and the use of shared memory for communication. Storing the entire set of powers may require up to $\sim$32.77 KB of shared memory, which can limit the maximum attainable occupancy. Therefore, there is merit in trying to reduce the amount of shared memory used by each thread block. This can be achieved by storing all the powers in per-thread registers. Inter-warp communication can then be accomplished with only one floating-point value per thread, requiring only $\sim$4.1 KB of shared memory per thread block. This form of communication requires additional synchronisation, since only one value can be exchanged per synchronisation, compared to up to eight values using the full shared memory approach.

GPU normalisation is implemented as an in-place CUDA kernel, which can make use of either the full or the limited shared memory median calculations described above. The kernel uses one thread block per segment of input data for each stack and is templated on the width of the segments. The choice of median selection is run-time configurable.

6.2.3.3 Multiplication

The multiplication component is well suited to many-core computation. It can be performed as a stand-alone CUDA kernel or as a cuFFT pre-transform callback. A number of alternative stand-alone CUDA kernels are discussed, as well as *slicing* – a generic modification to increase the number of active threads. The section concludes with the details of the pre-transform callback method.

The most naïve implementation of this large-scale pointwise multiplication would be to assign a thread to every point in each of the r-$\dot{r}$ planes. Each thread would read the relevant input and kernel values, perform the complex multiplication and write the result back to device memory. Due to the minimal amount of arithmetic, this is a memory-bound operation, thus, the focus of optimisation is memory access. The memory writes to the complex plane cannot be avoided, as each "output" value is unique. However, the two memory reads can be streamlined. Firstly, as is described in Section 5.3.2, all the planes of a stack share one convolution kernel. Thus, if the multiplication is performed on a stack rather than on an individual plane, one can drastically reduce memory accesses by reading each kernel value only once. Secondly, all the values in a column of an r-$\dot{r}$ plane use the same input value, therefore each input value need only be read once per plane. Thus, the optimal implementation need only read each input and kernel value once per stack, and write one complex value to memory for each element of a stack. Three different stand-alone CUDA kernels were implemented: `mult11`, `mult21`, and `mult22`. Each has a decomposition that suits a different combinations of parameters.

mult11

A CUDA kernel was implemented to perform the multiplications for a single plane. This kernel is referred to as `mult11` and requires a separate CUDA kernel to be launched for every plane. In this kernel, one thread is allocated to each column of the plane, reading the applicable input value and then looping down the column. Each iteration reads a kernel value, performs the complex multiplication and writes the result to the complex plane. Due to the layout and striding of the data, these memory accesses are aligned and coalesced, as is the case for all similar memory accesses in this section. This method minimises the number of times the input values are read and the number of registers used, however, it duplicates the reads of the kernel values.

mult21

The duplicated kernel reads of `mult11` can be minimised by having a single CUDA kernel perform the multiplication of an entire stack, rather than a plane. As before, the decomposition is one thread per column. Each thread handles multiple planes and, as such, reads all the relevant input values once and stores these in per-thread registers. The thread then iterates over a column of the convolution kernel. For each iteration, the kernel value is read and multiplied with the

applicable input values, and the result is written to the complex plane. This CUDA kernel is referred to as `mult21`, and is shown in algorithm 4. The drawback of this approach is that some stacks may contain many planes – up to 72 when $N_s = 9$. One complex input value for each plane is stored in registers; thus in some cases, each thread may require a very large number of registers to store these input values. Therefore, if there are a large number of planes in a stack, it may result in register spills that can drastically reduce performance, since the input values are used in the innermost loop.

mult22

An additional multiplication kernel that aims to reduce register use was subsequently implemented. This kernel is referred to as `mult22` and is given in algorithm 5. Here, the outermost loop iterates over the harmonics of the stack, requiring fewer input values to be stored in registers. This method does, however, require duplicating the kernel reads, once for every harmonic of the stack, of which there can be up to eight. The expectation is that these duplicated reads will have a relatively small impact, as they make up the smallest proportion of the memory transactions. Moreover, the kernel values read will often be cached. The `mult22` kernel is expected to perform well when there are many segments and few harmonics in a stack.

mult00

For the purposes of testing, an additional CUDA kernel – referred to as `mult00` – was implemented. This is simply an "optimal" reference CUDA kernel; it performs the minimum number of memory reads, writes, and arithmetic operations. The memory accesses are performed on the correct memory, and are all ideally aligned and coalesced. However, these are not in the correct order and there are no dependencies between them, meaning that the values written are incorrect. In addition, no values are stored in registers. Thus, this kernel can be used as a close-to-best-case run time for testing and comparison.

Slicing

In the kernels discussed above, there may not be enough threads to fully saturate a device for narrow plane width. In these kernels, one thread per column is used and the width of the stacks range from 2^6–2^{15}. To increase the number of threads preforming the multiplication, the plane

can be split into horizontal "slices". A thread is allocated to each slice of each column, a given thread will then only iterate over the $\dot{r}$ values in a slice. Using this method, the number of threads will scale directly with the number of slices. In a given column, all the input values will have to be read for each slice, duplicating some of the memory access. However, reading the input values accounts for the smallest proportion of the memory transactions; this can thus be expected to have minimal impact. Slicing the kernels will increase the number of thread blocks, which may allow hiding of some of the memory access latency. This sliced approach is applied to all the multiplication kernels discussed above. The degree of slicing is generally automatically determined at run-time, but can be manually specified.

Templating

To be able, at compile-time, to determine the number of registers required in these kernels, certain parameters – such as N_s and h– are required as hardcoded values. To achieve this, a hierarchical system of templated function calls is used to allow hardcoded combinations of all the various parameters. Having these values hardcoded, allows many compiler optimisations to be performed, which can drastically reduce the run time of the kernels. However, this form of nested templating can drastically increase the size of the compiled binary and the compilation time. To reduce the effect of this, the range of a number of the parameters can be restricted at compile-time. The main advantage of this compile-time configuration is that the reduced binary size will reduce CUDA context initialisation.

cuFFT callback

The other multiplication option is to use a cuFFT callback to do the multiplication as a pre-transform operation to the iFFT. To facilitate this, necessary information for each stack – such as N_s, the stride and memory addresses of the input and the kernel – are calculated and stored in constant memory during initialisation. The memory address of the applicable data structure is passed to each thread in the callback function. Prior to the transformation, the callback function is called for each point of the plane; each thread reads the input and the kernel values, performs the multiplication, and returns the result which is then used in the transform.

Algorithm 4 Stack Multiplication Kernel 1

```
1:  procedure MULT21
2:      for each harmonic h in the stack do
3:          for each segment s do
4:              Read the input value I_{h,s} into R_{h,s}
5:      for each y in the slice of the kernel column do
6:          Read the kernel value k_y
7:          for each harmonic h in the stack do
8:              if y is in the plane then
9:                  for each segment s do
10:                     Multiply R_{h,s} and k_y
11:                     Write result to device memory
```

Algorithm 5 Stack Multiplication Kernel 2

```
1:  procedure MULT22
2:      for each harmonic h in the stack do
3:          for each segment s do
4:              Read the input value I_{h,s} into register R_s
5:          for y in the slice of the harmonic do
6:              Read the kernel value k_y
7:              for each segment s do
8:                  Multiply R_s and k_y
9:                  Write result to device memory
```

Timing

Figure 6.5 shows the run times of the stand-alone multiplication kernels. The relative, rather than the absolute, run times are of interest for this analysis. The results shown here are the mean run times of the multiplication component of a single segment. The multiplication was performed using a single CG plan with: $Z_{\max}$ = 100, $\omega = 8192$ and $h = 16$ over a range of N_s. Each panel of the figure shows the run time for a single stack with the top being the largest stack. As is apparent in Figure 5.4, the number of harmonics per stack decreases in the smaller stacks. The number of planes in a stack is the product of h_s and N_s.

If one examines the top stack, one can see that the run time of the `mult00` kernels is very similar between the first two devices. This follows logically, since multiplication is a memory-bound operation and both devices have the same memory bandwidth. A run time of $300\,\mu$s results in an effective bandwidth of 162.31 GB/s, which is $\sim$72.36% of the theoretical bandwidth of 224.32 GB/s for the two devices. It is noted that the run time decreases significantly with stack size; thus the run time is dominated by the first stack.

In the new generation Maxwell card, the `mult21` kernel always performs best, and when N_s is large (≥ 6), this performance is very close to optimal. The superior performance of this kernel can be attributed to the large number of registers per thread (255) in the newer generation; while on the older Kepler card with fewer registers per thread (63), the run time of the `mult21` kernel clearly degrades beyond some point. It was found that if $h_s \times N_s > 25$ – at which point 50 registers would be used for the complex input values – the run time begins to increase. For a low N_s, the `mult22` kernel has a higher run time than the `mult21` kernel, but from the point at which latter kernel begins to spill registers, the former begins to perform close to optimally.

As anticipated, the per-plane multiplication (`mult11`) is largely unaffected by N_s, and has a slower run time than the stack multiplication kernels. Increasing N_s improves the run times of the per-stack multiplications, and for larger N_s one can obtain close-to-optimal run times on both generations of GPUs.

The other option is to do the multiplication as a pre-iFFT callback. Theoretically, this should be the best of the options, as it could almost completely remove the memory-bound multiplication by incorporating it into the unavoidable memory read of the iFFT. However, it was found that doing

the multiplication in a callback is approximately seven times slower than the best stand-alone kernel. On investigation, it was found that simply reading the correct input from the complex plane in a callback – which requires no computation and is theoretically equivalent to the standard iFFT with no callback – takes more additional time than running the entire stand-alone multiplication kernel. It is not certain why this is so, but it is assumed that it is due to some synchronisation inefficiency in the pre-transform callback, implemented in the current cuFFT library [100]. Thus, using a pre-transform callback is not currently viable. This option is theoretically the most efficient and may become feasible in future versions of the cuFFT library [100]; it is thus still included in the code but compile-time disabled by default.

6.2.3.4 Fourier transforms

The cuFFT library [100] is used to perform the FFTs and iFFTs on the device. All transforms are performed on a stack's-worth of data as an asynchronous, batched transform. Each batched transform requires a cuFFT plan and these are created during the CG initialisation, and form part of the CG plan. These cuFFT plans allocate temporary work areas which generally require the same amount of memory as the input of the transform. This additional memory may become significant as more CG plans are created. Large batched FFTs, such as the ones performed here, often fully utilise the device, meaning that there may be little to no concurrent kernel execution occurring while these large batched transforms are running. Thus, it may be assumed that these batched transforms will generally not run concurrently, in which case they can use the same cuFFT plan, resulting in less temporary memory being required. Therefore, the option of creating only one cuFFT plan per stack, per device was allowed in this work. When using a single cuFFT plan, all cuFFT calls need to be made in a thread-safe manner from the host code, thus they are enclosed in an OpenMP critical block. On the GPU, the requirement that no similar cuFFT kernels may run concurrently is enforced by running all related transforms in a single CUDA stream. This serialises the cuFFT kernels with respect to each other, even when launched asynchronously from separate CPU threads.

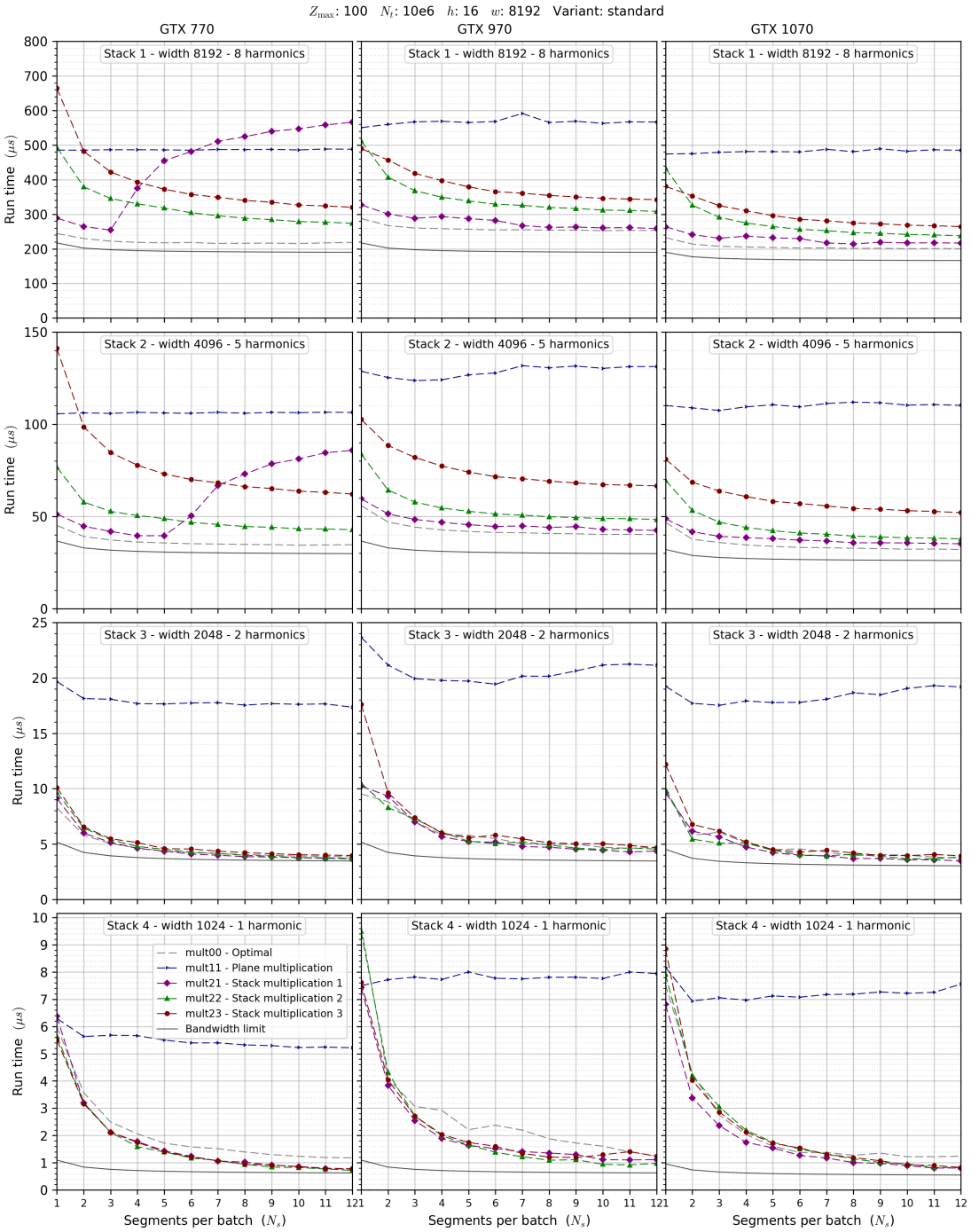


Figure 6.5: The run time of the GPU multiplication component across GPUs (columns) and stacks of a batch (rows), shown across a range of N_s. These are normalised for a single segment. On the newer GPUs, the `mult21` kernel generally has the best performance across the full range. The steep rise in the run time of the `mult21` kernel on the older GTX 770 is caused by register spilling; however, from the point at which this becomes problematic, the `mult22` kernel can be used. For the larger stacks, these kernels perform close to optimally, with performance increasing as N_s is increased. The per-plane multiplication kernel (`mult11`) is much slower and is included to illustrate the value of aggregating planes into stacks.

6.2.3.5 Power calculation

The power calculation is performed as a post-iFFT callback. The primary advantage of this approach is the reduction in the amount of data to be stored by the iFFT and read by the SAS components. The callback function receives the post-transform complex values, calculates the power, converts it to the appropriate data type and writes it to the appropriate memory.

In the standard variant, the callback simply writes the powers to the plan-specific powers plane. The powers plane is the same width as the complex plane; allowing the callback to simply use the iFFT-specific offset of each point – passed to the callback function as a parameter – to write its output to the powers plane. Using a callback has some overhead which increases the run time of a transform. We find that using a post-transform callback that simply writes the complex results to the correct location, can increase the run time of a large batched iFFT by up to 35%. This increase in the run time is particularly noticeable in the older Kepler GPUs. The overhead can be offset by the significant reduction in memory written when storing half-precision powers. Using a post-transform callback to store half-precision powers on a large batched iFFT changes the run times by 1.2, 0.84, and 0.75 times those of the base transform with no callbacks, for the GTX 770, GTX 970, and GTX 1070 respectively. Thus, this callback can reduce the run time of the transform on newer generation GPUs. The 20% increase in run time on the older Kepler GPU is still less than the time required to calculate the powers by alternative methods. Consequently, performing the power calculation as a post-transform callback is still the most efficient alternative.

The in-memory pipeline is slightly more complicated as the callback has to write the results to the full r-$\dot{r}$ plane. This requires the callback to translate the iFFT-specific offset to the relevant offset in the in-memory plane. The contamination complicates this translation, since only a subset of the iFFT results should be written to this plane. When using a batched transform operating on a stack containing multiple planes, this translation requires five coefficients[2] to compute. It was found that the overhead added by accessing these values, even from fast constant memory, results in a significant increase in the run time of the iFFT. To overcome this, we use a novel technique, encoding the five values in a single 64-bit value. This can then be passed to the callback function masked as a device pointer[3], which is a very efficient means of passing the coefficients to this

[2]These include run-time configuration parameters such as the width, amount of contamination, and the stride of the full r-$\dot{r}$ plane.

callback. The run times of our in-memory transforms are slightly higher than their standard equivalents; this is due both to the time required to access the pointer parameter, and the memory writes to the full r-$\dot{r}$ plane being more disparate than those to the powers plane.

6.2.3.6 Sum and search

We combine harmonic summing and searching into a single SAS kernel, so that the intermediate values need not be stored in device memory. As with `Accelsearch`, the resolution in both r and $\dot{r}$ are the same in all the r-$\dot{r}$ planes of a family. Thus, when summing points to the primary plane, the actual resolution of locations required from the smaller sub-planes is higher than the resolution at which they are sampled. This means that some approximation is performed by which the closest calculated point to the actual location required is used in the summing. A consequence of this is that when doing the summing, smaller sub-planes have an effectively lower resolution. Thus, harmonic summing can be thought of as "stretching" each sub-plane to the same size as the primary and then adding them. This method has the advantage that higher harmonics have a relatively finer resolution, giving better localisation. This does, however, introduce some inaccuracy in the summed powers, which is corrected for in candidate optimisation. The basic structure of the SAS kernel is given in Algorithm 6 and includes a number of features, each of which is discussed below. As with the multiplication kernels, a hierarchical system of templated function calls is used to allow hardcoding of a number of the key parameters and data types.

The SAS kernels differ between the standard and in-memory implementations. In this work, the in-memory search is only performed once the entire in-memory r-$\dot{r}$ plane has been generated. This is performed in an iterative manner, looping over segments of the full r-$\dot{r}$ plane and using these as the primary plane in the staged harmonic summing procedure. The width of these sections can be significantly wider than the plane widths used in the standard search. This negates the need for the in-memory CG plans to handling multiple segments or use slicing in the in-memory SAS kernels, making it slightly simpler than the kernel used in the standard search.

The SAS kernel has a minimal amount of computation for each memory access; these computations comprise: indexing, summing and keeping track of maxima. Accordingly, this kernel could be considered memory-bound, however, there are a number of factors that shift the kernel away

[3] This is the only user-defined parameter passed to the callback function.

from being truly memory-bound. The principal factor is that the memory reads are not independent – the powers are added to a running sum, and this is in turn used to maintain a running maximum. These dependencies between consecutive reads mean that fewer memory transactions can be initiated before having to wait for the result of previous transactions, thus reducing the effective bandwidth.

Registers

Broadly speaking, candidates are unique in r, thus it is logical for each column of a primary plane to be searched by one CUDA thread. Each thread need only return the maximum in the column, for each stage of harmonic summing. Therefore, each thread requires a number of registers to keep track of the $\dot{r}$ location and value of the maximum for each segment. Increasing N_s will thus increases register pressure, and may cause detrimental register spilling beyond some size.

Coalescing

As the values of each harmonic are summed, the threads of a warp read increasingly smaller sections of the powers plane, resulting in multiple threads reading the same memory locations. Due to the layout of the planes in memory, these memory accesses are coalesced. Similarly, if one considers a single thread looping down a column of the primary plane, successive rows of the primary plane may access the same row in the sub-plane due to the differing resolution. Thus, the last item accessed in each sub-plane is stored in a register, effectively resulting in each element of the various sub-planes being read only once, decreasing memory bandwidth. This single-read pattern removes the need to use shared memory to cache values.

Chunking

It was found that the efficiency of the SAS kernel can be increased by reading *chunks* of powers into registers and then processing them, essentially separating memory access and computation into a number of small sections. The "search" can then be carried out using the powers stored in these registers, rather than iterating over individual powers in device memory. This allows the memory reads of a chunk of powers to be processed together, which can increase effective bandwidth and reduce instruction dependencies in the innermost loop. This approach comes at the cost of additional register use, as registers are needed to store the powers for each segment.

Once more, this increased register pressure may cause register spilling if the number of elements in a chunk (N_c) is too large.

Slicing

In the standard search, the width of the primary plane ranges from 1024 to 32768. As with the multiplication kernels, having a thread per column means there may not be sufficient threads to saturate the device, thus decreasing occupancy. A similar approach of dividing the primary plane into a number of horizontal *slices* was therefore implemented. This approach increases the amount of output memory required by a batch and can potentially increase the number of candidates that need to be processed by the candidate storage component, both of which may affect the speed of the search.

Indexing - constant memory

For each harmonic of a family, each column of the primary plane will access the same x offset in the sub-plane, and each row of the primary will access a specific row in the sub-plane. If the start of each segment is aligned on a multiple of h, then the x indexing of a given column will be the same for all segments in the batch. The x indices for each sub-plane are thread-specific and thus need only be calculated once by each thread. The y indices are specific to the row and sub-plane, as a result they are the same for each thread and indeed all segments of the search. These y indices are needed in the inner most loop, and are common to all threads. Therefore, these values are precalculated during CG initialisation and stored in constant memory, allowing the values to be easily and efficiently accessed by each thread during the inner levels of iteration.

Candidate writes

Each thread need only return a candidate if the maximum summed power is above the relevant detection threshold. Making this memory write optional has the advantage that if no threads of a warp find valid candidates – which is the typical occurrence – this memory transaction can be skipped. To ensure that there are no "junk" values in the output memory, each thread zeroes the output memory specific to it, early in the execution of the kernel.

In-kernel candidate counting

We use an additional feature in which the total number of candidates found in a batch can be counted and written to device memory. This requires an atomic addition which can slow the SAS kernel on older GPUs. However, if no candidates are found in a specific batch, simply checking this sum can negate the need to run the following component, potentially saving some host-side computation.

Algorithm 6 Sum & Search Kernel

```
1:  procedure SUM & SEARCH
2:      for each harmonic h do                                  ▷ Calculate x indices
3:          Calculate x_h the x offset for that sub-plane
4:      for each stage a do                                     ▷ Zero candidate storage
5:          for each segment s do
6:              Zero candidate memory
7:
8:      for each chunk c in the slice of the plane do
9:          for each y in chunk do                              ▷ Zero chunk powers
10:             for each segment s do
11:                 P_{s,y} = 0
12:         for each stage a do
13:             for each harmonic h do
14:                 for each y in chunk do
15:                     y_f = c * chunk length + y
16:                     for each segment s do                   ▷ Accumulate chunk powers
17:                         Read y_{h,y_f} the sub-plane row index from constant memory
18:                         Read S_{x_h,y_{h,y_f}} the power from sub-plane
19:                         P_{s,y} = P_{s,y} + S_{x_h,y_{h,y_f}}
20:             for each segment s do                           ▷ Maximum comparisons
21:                 for each y in chunk do
22:                     if P_{s,y} > Max_{a,s} then
23:                         Max_{a,s} = P_{s,y}
24:
25:     for each stage a do                                     ▷ Candidate storage
26:         for each segment s do
27:             if Max_{a,s} > Cutoff_a then
28:                 Write Max_{a,s} to memory
```

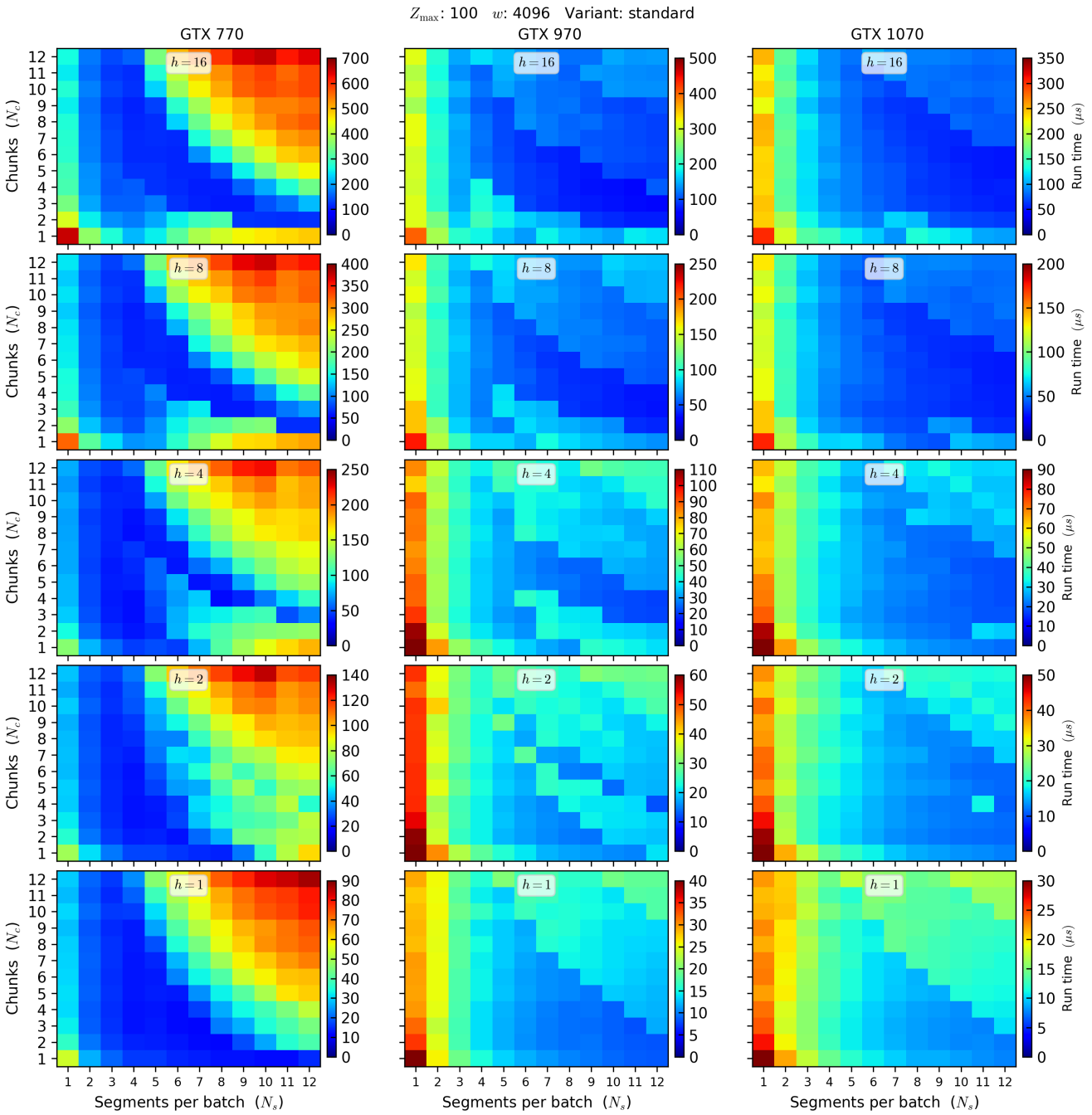


Figure 6.6: The run time (µs) of the standard variant of the SAS component of a CG plan with $Z_{\max} = 100$ and $\omega = 4096$, normalised for a single segment and using the default number of slices. The GPUs used are shown in columns and the number of harmonics summed are shown in rows. Increasing both N_s and N_c can decrease the run time. However, both parameters increase register use, and at some point this begins to degrade performance. This is particularly pronounced on the GTX 770 which has fewer registers than newer generations of GPUs. The optimal combination of N_s and N_c can be seen to vary both with the GPU architecture and h. The multiplicative nature of the number of registers used results in the hyperbolic boundaries visible in the figure.

Timing

As with the `mult21` kernel, one of the key factors that can degrade the performance of the SAS kernel is register spills. The number of registers required by this kernel scales with N_s, N_c, and h. Thus, care must be taken with the configuration parameters selected so as to avoid register spills wherever possible.

The N_s parameter is common to all of the components of a CG plan, while N_c is specific to the SAS kernel. Thus, in this analysis, N_s and h are assumed, and an attempt is made to identify the optimal N_c for the combination of the two. The run time of the SAS kernel is shown in Figure 6.6. The combination of N_s and N_c has a significant impact on the run time of the kernel, with bad combinations doubling or tripling the run time. As with the `mult21` kernel, the main cause of this degradation is register spilling. The number of registers used is proportional to N_s and N_c, which leads to the hyperbolic boundaries visible in the figure. These boundaries vary across h and the generations of GPUs. The older Kepler card has fewer registers per thread which makes the performance degradation more severe. Some changes with other parameters have been observed, in particular: ω, $Z_{\max}$, and number of slices, although these are generally minor.

The combination of $N_s = 1$ and $N_c = 1$ (ie no chunking and not aggregating segments) is very slow, showing the value of both techniques. The optimum combination is generally $4 \leq N_s \leq 8$, with the optimal N_c varying with h. An attempt was made to find an analytical relationship, such as was found with the multiplication kernels, but no such relationship was found. Thus, the optimal N_c has been precalculated for a broad range of parameters, and hardcoded. However, these defaults can be overridden with a run-time . There is scope to develop an auto-tune feature that can determine these optimal values of N_c, as well as many other configuration parameters for the specific environment on which the application is run.

With the optimal combination of parameters, the effective bandwidth of our SAS component is between 55% and 30% of the theoretical bandwidth. Summing only one harmonic results in the highest bandwidth (55–45%), due to the simplicity of the implementation. For all other stages of harmonic summing, the effective bandwidth is between 40% and 30% of the theoretical maximum, with the best performance generally being found with large values of ω and N_s.

6.2.3.7 Candidate storage

In `Accelsearch`, the initial candidates are stored in a singly linked list. The access to this list was modified to be thread safe and was implemented in the GPU search.

To more easily store and access the initial candidates found during the CG stage, an additional candidate storage method was implemented. This is simply an array of candidate objects, with one location for each possible candidate r value. The actual candidate data type consists of the power, sigma, number of harmonics summed, r, and $\dot{r}$. This method has the advantage that all the required memory is allocated during CG initialisation and that access time is very fast. However, the amount of memory required to store the array can be prohibitive for long observations.

6.2.4 High-level parallelism in candidate generation

The previous section described the CUDA kernels used in the CG stage; these kernels exploit the lowest level of task parallelism, and are expected to yield the greatest performance gains. There are, however, higher levels where parallelism can be leveraged to increase performance. A number of the tasks performed in the CG initialisation, such as calculating the convolution kernels, can be performed concurrently with other initialisation tasks. In the CG stage, multiple CG plans are used to process the iterations of the main loops of the CG stage. These plans are independent, allowing the logic and control of each to be run in a separate CPU thread. The processing of a plan is pipelined so that a single plan data structure contains data from multiple iterations, and allows components of these to run concurrently. These components comprise CPU computation, GPU computation, and data transfers to or from the device.

6.2.4.1 Initialisation

For each GPU to be used in the search, a CUDA context needs to be initialised. This initialisation would usually occur in the background when the first CUDA function requiring a context is called. Context initialisation can take some time – up to a second, depending on the device, binary size, and system configuration. To speed up the search, a separate Portable Operating System Interface (POSIX) thread is spawned to initialise a CUDA context for each device. This is achieved by allocating a temporary memory value, which in turn is used to determine the size

of memory striding for each device. These threads are launched as early as possible, prior to CG initialisation, allowing the parent thread to continue with other potentially time-consuming search preparations such as reading the input DFT into memory. The CUDA context thread is joined to the main thread during CG initialisation, prior to the point at which device memory is allocated.

The convolution kernels are calculated on the GPU during the CG initialisation. This is done asynchronously on a GPU, allowing the CPU to continue with other initialisation tasks. These tasks include a number of GPU memory allocations for the plan data structures. Any device memory allocation will force synchronisation of the relevant GPU, thus limiting the degree of asynchronous GPU computation during initialisation. However, the time taken for this initialisation is usually trivial.

6.2.4.2 Concurrent plans

To allow multiple plans to be processed concurrently, the logic and control of each is run in a separate CPU thread. These CPU threads are implemented as an OpenMP parallel block enclosing each of the main loops in the CG stage. Each thread is assigned a single preinitialised CG plan data structure, with associated host and device memory. The CPU threads can operate relatively independently, with CUDA kernels from the separate threads able to run concurrently with other kernels, as well as CPU computation.

6.2.4.3 Pipelining

In some situations, it may be beneficial or necessary to use fewer plans, and thus fewer CPU threads, to process the main loops of the CG stage. A single thread processes multiple batches using a single plan data structure. Within a single CPU thread, a degree of concurrent execution can be achieved by pipelining the processing of multiple batches. As mentioned in Section 5.3.1.2, asynchronous GPU execution can be used to allow four-way concurrency with respect to a single CPU thread.

The processing of each batch can be broken down into nine components: input preparation, input normalisation, copy input H2D, FFT, multiplication, iFFT, SAS, copy results device-to-host (D2H), and candidate storage. The reader is reminded that the input normalisation and FFT components can be performed by either the CPU or the GPU, thus the point at which the H2D

memory transfer is performed is dependent on the configuration of a specific search. These components operate on six plan-specific memory spaces: the two pairs of input and output memories, the complex plane and the powers plane. The data from a single batch can be thought of as moving consecutively from one memory space to another, with some data transformation or processing occuring between each. Thus, the memory spaces can be used to demarcate the nine components into eight *units*, where each unit is made up of one or more components that operate on either one or two of the memory spaces. These units are shown as the rows in Figure 6.7a and comprise two sections of CPU computation, four of GPU computation and two memory transfers.

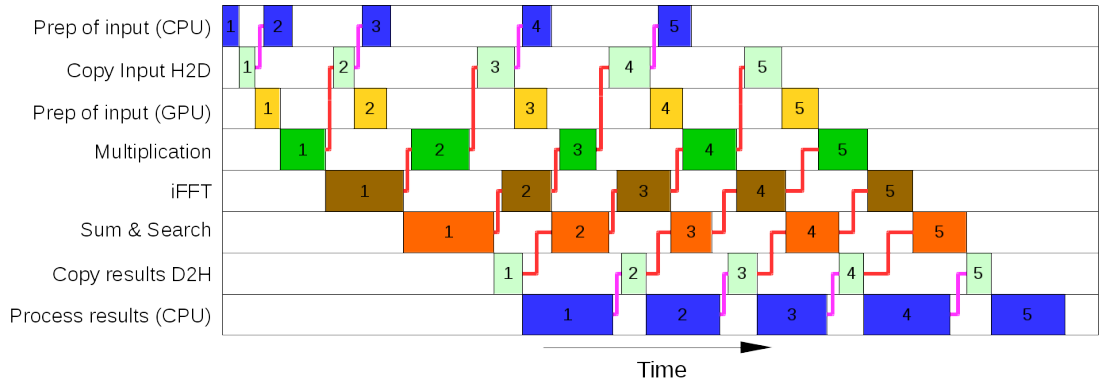
To process a single batch, these units need to run consecutively. However, when considering separate batches, any pair of units that do not operate on a common memory space may be performed concurrently. For instance, as is shown in Figure 6.7a, the multiplication component of one batch can be performed concurrently with the SAS component of the previous batch. This is due to the fact that multiplication reads data from the input memory and writes to the complex plane, while the SAS component reads from the powers plane and writes to the output memory. Accordingly, the six memory spaces mark six synchronisation points between the units of consecutive batches. These synchronisation points are shown as the stepped vertical lines in Figure 6.7a. The red lines represent non-blocking synchronisation, while the magenta lines show the two blocking synchronisations corresponding to the transfer of data to and from the device.

To achieve the desired pipelined execution, the kernels and memory copies are launched asynchronously and are not called in their order of execution. This requires each iteration to launches a number of kernels and memory transfers that operate on data from a range of iterations. CUDA streams, CUDA events, and explicit stream synchronisations are used to ensure that the correct execution order is maintained.

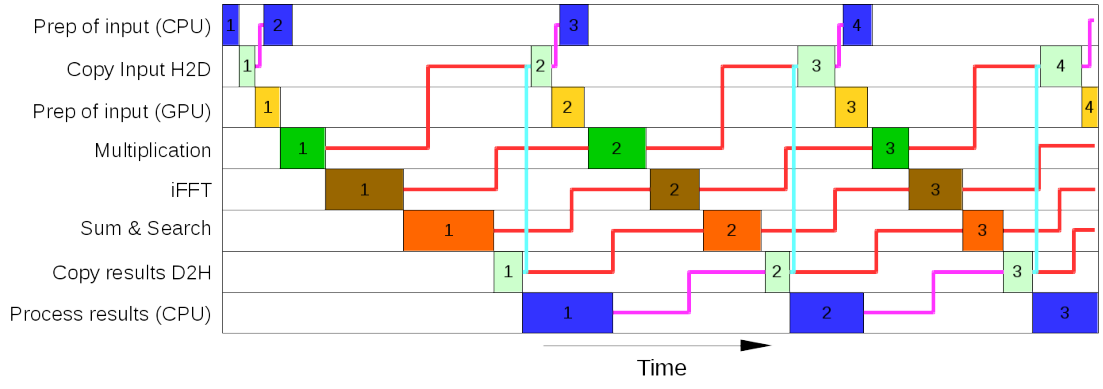
With this scheme, the first iteration will not call all the components associated with the relevant batch. Instead, during the first iteration, the input data of this batch is processed by the CPU. This is followed by the asynchronous launch of the H2D memory copy and a number of the CUDA kernels. The second iteration will start in a similar fashion, with the CPU processing the input of the second batch. Thereafter, it asynchronously queues up the H2D memory copy and some of the CUDA kernels that operate on the data of the second batch. Only once these have been queued, the D2H memory copy that deals with the data of the first batch is asynchronously called. The

third iteration initially processes the input of the third batch and queues up the applicable CUDA kernels, only then is there a blocking synchronisation after which the CPU processes the results of the first batch. Once this is complete, the H2D memory copy of the data of the second batch is asynchronous launched. This iteration is the first to run all nine components, all subsequent iterations repeat the same steps and result in the complete processing of a single batch.

This system allows a single CPU thread to pipeline the processing of multiple batches. For example, in the third iteration, the GPU can be running CUDA kernels launched asynchronously during the first and second iterations, while the CPU processes the input of the third batch. Thereafter, the CPU processes the results of the first batch; while this is being performed, the H2D memory copy of the third batch can be run, as can the CUDA kernels operating on data from the second and third batches. In situations where the amount of CPU and GPU computation are similar, this type of asynchronous pipelining can half the run time of the main loops of the CG stage.



(a) Out-of-order calls



(b) In-order calls

Figure 6.7: The asynchronous pipeline of a single CG plan. When the components are launched asynchronously but in order, the execution is serialised by the dependency between the H2D and D2H memory copies. Launching the components out of order can resolve this.

Candidate storage threads

The final component of the CG stage – that of processing the results – is performed by the CPU. This component incorporates the computationally intensive task of calculating the sigma values for the initial candidates. Naturally, this is only done if the value of the summed powers is above the relevant detection threshold. It was found that these calculations can take a nontrivial amount of time if there are many candidates found in an iteration; in some cases, twice to three times longer than the time required for all other components of that iteration. This is often the case if there is significant RFI present, or if the spectrum has strong red noise. The presence of red noise often results in a number of the low-frequency iterations generating a large number of initial candidates. Processing these candidates in the main CPU thread could delay the processing of the subsequent iterations if the CPU computation takes longer than the queued GPU tasks.

To overcome this, a separate CPU thread is spawned at the end of each iteration to perform the candidate storage component. The majority of these threads will have little to do, and will terminate after minimal time, while a small number will continue to process for significantly longer than the mean time taken to process one batch. The time required to spawn these threads is significantly less than the run time of the other computation in the iteration, thus the use of a pool of preallocated threads was deemed unnecessary. Data is passed to a child thread by allocating a temporary host memory location and copying the values from the batch's pinned results memory to this temporary location. Thereafter, the address of this new memory is passed to the newly spawned thread. No synchronisation is needed between the parent and child threads once the child has been launched, thus these threads can run unhindered in the background while the parent continues to iterate through the batches. The input DFT is searched from low to high frequency, this allows the majority of threads with red noise candidates to be spawned early in the search, and as a result they will have the most time to complete. In all cases, there is only one global data structure to hold all the initial candidates. Therefore, adding candidates to this data structure is performed in a thread-safe manner. The threads are created using the POSIX threading library. Thread-safe memory writes are done using a POSIX mutex lock, while a semaphore is used to ensure that all threads have completed before moving on to the CO stage.

In-memory

In the in-memory variant, iterations of the plane generation sub-stage can overlap in a similar fashion to the iterations of the standard search, as the tasks consist of multiple components. The search sub-stage consists of only one GPU component, one CPU component and one memory copy. If the SAS kernel and memory copy are to run concurrently, the SAS kernel cannot write to the same memory that is being copied to asynchronously. Thus, two output memory locations are needed to allow each successive iteration to copy and write to different memory in a flip-flop manner. In the search sub-stage, the complex plane is re-purposed and used as one of these memory locations. Again, processing the results is done by the CPU, which can optionally be performed in a separate thread for each iteration, in the same manner as the standard search variant.

6.2.4.4 Synchronous mode

The synchronous execution mode was specifically implemented to allow debugging and timing of the various components of the GPU search (Section 5.3.1.2). The control of synchronous execution has some associated overhead. To prevent this from impacting performance-critical execution, all the operations specific to the synchronous mode can be removed at compile-time with a preprocessor macro. The actual decision of whether or not to run in synchronous mode is specified by a configuration parameter in the text configuration file.

To achieve synchronous execution on the host, the main loops of the CG and CO stages are limited to a single CPU thread. This thread still uses multiple plan data structures, depending on the iteration. The CPU components that are usually run in separate POSIX threads, such as candidate storage, are still run as separate threads; however, the main thread is blocked and waits for the completion of the child thread before continuing. Thus, all CPU control and computation is run synchronously, using the standard asynchronous data structures and methods, allowing more accurate timing of various CPU components.

The synchronous and asynchronous modes are similar in that a number of CUDA kernels are asynchronously queued up in the standard CUDA streams. However, in synchronous mode, a number of additional events are added. These additional events are used to explicitly serialise and

time the execution of the various CUDA kernels and memory copies.

The copies between the device and host are asynchronously queued at the same time as the CUDA kernels. These are queued in such a way that they execute asynchronously with respect to the CPU and GPU computation. Thus, the synchronous execution mode is not truly synchronous; it exhibits three-way concurrency with CPU computation, GPU computation, and memory copies all occurring simultaneously. These three can all operate independently without affecting the timing of the others, allowing accurate timing of the individual components while, at the same time, reducing the run time.

6.3 Candidate optimisation implementation

The CO stage takes the initial candidates found by the CG stage and removes spatially and harmonically related candidates, usually greatly reducing the total number. The r-$\dot{r}$ location of each of the remaining candidates is then refined and, finally, a number of metrics are calculated for reporting. The location refinement can be performed by either the CPU or GPU alone but is generally done by both, with the GPU performing an initial, large-scale localisation followed by high-accuracy, fine-scale localisation by the CPU. The metric calculations are performed solely by the CPU.

6.3.1 High-level parallelism in candidate optimisation

The number of initial candidates a search produces can vary and is often affected by the amount of periodic RFI. Each candidate can be optimised independently which allows multiple candidates to be concurrently optimised. To facilitate this highest level of parallelism, an OpenMP parallel block is used around the main loop that iterates over the candidates. The number of threads used in this block (N_o) is configurable, but it was discovered that fewer threads – usually in the range of four – are often the fastest. Each candidate optimisation generally has similar resource requirements; therefore, in a similar manner to the CG plan data structures, each CPU thread can use a CO plan data structure containing a single set of resources to optimise all of its candidates. These resources are allocated in CO initialisation, and each thread of the main optimisation loop is allocated a CO plan data structure encompassing a number of CPU and GPU resources.

The optimisation of a single candidate by a single CPU thread is now examined. The GPU r-$\dot{r}$ location refinement is achieved by iteratively creating sections of harmonically summed r-$\dot{r}$ plane. This second level of iteration discussed here is not to be confused with the higher-level iteration over the individual candidates. A single iteration requires calculating a grid of harmonically summed r-$\dot{r}$ points, and then locating the maximum power in this grid. These grids are calculated using the `rr_plane` function (Section 6.1.2). Each harmonic summed requires a normalised section of the input DFT. The first iteration normalises and copies these values to the device. The size of each input section is inflated to be larger than the minimum width required for the first plane. Thus, if future iterations move the location of the candidate by only a small amount, the same sections of input can be used, negating the need to normalise and copy input data to the device for each iteration. Once the r-$\dot{r}$ values have been calculated by the GPU kernel, the entire plane is copied back to the host, as the results and logic of the dimensions and position of the next grid to be created are handled by the CPU. This dependence means that the GPU computation and memory copies must run sequentially. The amount of data transferred is usually low, thus the GPU computation is generally longer than the memory transfers. It is noted that allowing multiple candidates to be concurrently optimised by separate CPU threads allows the computation and memory transfers of the different threads to run simultaneously.

Once a location has been adequately refined on the GPU, an additional location refinement and metric calculation needs to be performed by the CPU. The aim is to balance the computation so that the GPU and CPU components run in a comparable amount of time. Both components, however, cannot run in the single CPU thread, as the GPU refinement process is iterative, and requires regular blocking synchronisation between each iteration. Thus, the controlling CPU thread may be predominantly idle during the GPU location refinement, although it has infrequent short periods of activity during this time. In a similar fashion to the processing of initial candidates in the CG stage (Section 6.2.4.3), a separate POSIX thread is spawned to do the final CPU calculations once the GPU location refinement is complete. This can potentially create a large number of CPU threads: one per candidate. Each thread, however, is expected to have a fairly short lifespan. The rate at which these threads are spawned should be similar to the rate at which they terminate, with only a few persisting for an extended period. This means that at any one time there should not be an excessively large number of active CPU threads. These threads allow CPU computation to be performed concurrently with the GPU computation and memory transfers.

6.3.2 Optimisation memory

The optimisation of a single candidate performed using a CO plan. The GPU optimisation component creates increasingly finer resolution grids of harmonically summed r-$\dot{r}$ values; for this, a section of input data for each harmonic is required, as well as a buffer to store the grid. Accordingly a CO plan has two sections of paired pinned host and device memory: one for the input and one for the output grid. These CO plan are allocated at the beginning of the optimisation stage, and require significantly less memory than the CG plans of the CG stage. The latter are deallocated just before the optimisation stage, ensuring that there will be sufficient device memory available.

6.3.3 Components

6.3.3.1 GPU location refinement

The location refinement component comprises a mix of GPU and CPU computation. The GPU computation makes use of the `rr_plane` function (Section 6.1.2) to generate sections of r-$\dot{r}$ plane. This GPU computation will be discussed here, focusing on the main parameters of the GPU location refinement component.

The GPU location refinement component is an iterative process with each iteration creating a harmonically summed section of r-$\dot{r}$ plane. The notion of *levels* is introduced into this iterative process, with each level having planes of similar size and resolution but differing in location. Each time the size of the plane is reduced, the level is increased. The main considerations of this component are the number of levels and, for each: the size, dimension, precision, and accuracy of the section of plane created. Thus, the parameters are per-level configurable.

The aim of the first and, potentially, the second level of planes is to identify the correct maximum to refine. Thus, size of the first planes should be larger than the mean size of the features in the r-$\dot{r}$ plane, so as to encompass all the significant maxima of a feature. Furthermore, the resolution should be high enough to adequately resolve the separate maxima. The number of harmonics summed plays a role in the resolution as well. Higher harmonics will have larger planes, and the resolution in these must be high enough to resolve features. Therefore, a higher resolution

is required when a larger number of harmonics are summed. Due to the size and resolution requirements of these initial planes, a fairly high dimension is required. This large dimension can drastically increase computation. However, these planes can be calculated using the very efficient `pln_brd` kernel and need only be calculated at single-precision using the shorter, standard-accuracy filters.

The aim of the later levels is to refine the maximum at high resolution. These planes will be significantly smaller than a single DFT bin, and will generally only cover a small section of a relatively smooth r-$\dot{r}$ surface. In this regime, a higher resolution can be achieved by two methods. The first increases the resolution of the individual planes, and the second includes more levels. Including more levels with low-dimension planes is more computationally efficient, as computation is quadratically proportional to dimension (Section 6.1.2). Thus, square planes of dimension 32 are used for these levels, as anything less would be faster but less efficient (Section 6.1.2). These calculations should be done with the longer high-accuracy filters.

The grid parameters are per-level configurable. The dimension of each level was chosen as the primary parameter, as this has the largest effect on computation. To get the desired resolution in the first level, the size of the plane is scaled by the number of harmonics summed, with the size of the planes of each successive level being a set factor of the previous one. If this scaling factor is too large, each level may require multiple iterations to locate the maximum. If the scaling factor is too small, the resolution will not increase particularly fast, requiring additional levels. The ideal scenario is a single iteration per level. From inspection, it was found that a scaling factor as high as ten works well if the maximum location is taken as a weighted mean over a number of the largest points, rather than the location of the single largest value.

It was established that the use of double-precision planes in the GPU location refinement component has the largest impact on speed. In Section 6.1.2, it was shown that double-precision plane calculations are approximately 40 times slower than the single-precision equivalents. Accordingly, it was found that it is usually faster to do all the double-precision refinement on the CPU, as this can be done concurrently with other GPU computation. Having said this, if the location is not adequately refined in the initial single-precision GPU refinement, the time taken for the CPU computation can begin to significantly limit performance.

Table 6.1 shows the default per-level parameter selection for optimisation, with a scaling factor

Table 6.1: The default configuration parameters of the GPU location refinement component. The "Dim" is the dimension of the planes. The first level has a large size and dimension so as to resolve larger features, while the lower levels have a smaller size and dimension, and are intended to resolve the maximum at high resolution.

Level	**Dim**	**Accuracy**	**Precision**	**Resolution**				
				1 harm	**2 harm**	**4 harm**	**8 harm**	**16 harm**
1	128	Standard	Single	1.40e-2	1.22e-2	1.05e-2	8.75e-3	7.00e-3
2	64	Standard	Single	2.57e-3	2.24e-3	1.92e-3	1.60e-3	1.28e-3
3	32	High	Single	5.17e-4	4.52e-4	3.87e-4	3.23e-4	2.58e-4
4	32	High	Single	5.16e-5	4.52e-5	3.87e-5	3.23e-5	2.58e-5

of ten, and initial plane sizes of 16, 14, 12, 10 and 8 for the respective summing of 1, 2, 4, 8 and 16 harmonics. The table includes the resolution for each harmonic at each level. With these defaults, no double-precision calculations are done on the GPU, and the location is resolved to a resolution of $\sim$5e-5 Fourier bins.

6.3.4 Validation

To examine the validity of the new location refinement method, we compare the final r-$\dot{r}$ location and associated summed power of an initial candidate, optimised with the old and new methods. As discussed above, the new method has a number of configuration parameters, each of which may affect the accuracy result. A full analysis of the effects of all the combination of parameters is not included, as this would be too lengthy to report here. Rather, only the default parameter combination as outlined above is examined. This combination of parameters was specifically chosen to balance accuracy and performance, as well as CPU and GPU computation.

Each initial candidate is unique, prohibiting a generic validation. A single case study with a large number of typical real-world candidates will thus be examined. The results of a search of a single data set will be shown; this data was chosen as it generates a large number of initial candidates and contains a number of pulsars as well as a moderate amount of periodic RFI. From observations made, these results are very typical and the analysis below can be extended to most searches. For each candidate the two location refinement methods were run, using identical input. The sigma value of the final power is used as a yard stick, as the actual power is relative to the number of harmonics summed, and the candidates include a range of harmonics from one to sixteen.

Figure 6.8 shows the difference in sigma and euclidean distance, in Fourier bins, between the final maxima of the two methods. Each candidate is shown as a circle, with the size of the circle indicating the magnitude of the final GPU sigma value, thus larger circles indicate stronger candidates. The results are shown on a log-log plot with the vertical axis indicating the magnitude of the change in sigma, and the colour of the circles being used to indicate the sign of this change. The green and blue circles show where the new method resulted in a larger sigma value, indicating a higher maximum, while the red and magenta points represent an inferior maximum.

The results can be categorised into three main classes. The first class are those in which the change in σ and distance are very small, thus the final results may be considered practically equivalent. These make up the largest proportion of the results and can be seen as the large cluster of candidates in the lower left corner of Figure 6.8. Approximately 25% of the candidates have a sigma change of zero and thus cannot be shown in the figure, but are included in this population of similar results. These similar candidates make up the vast majority (75%–80%) of the results and are of little concern as they are considered to be practically identical. It is noted that even though these candidates may be considered to be effectively the same, the new method generally outperforms the old method, finding a similar location but with a slightly larger power. In the particular case shown, the new method produced a better maximum for all of these candidates.

The second class of results are to be found in the top right of Figure 6.8. These results represent the cases where the two methods find maxima that are far apart and have significantly different powers. These have a large change in sigma and thus have to be associated with candidates that have a large initial, or final, sigma value. It is evident that in this case, these are all relatively large circles, showing that the new method found a significantly higher maximum some distance from the location of the initial candidate. This is almost always due to the initial candidate being found as some small but significant maximum that is located in an artefact in the r-$\dot{r}$ plane caused by a very strong primary candidate. These artefacts are generally to be found in the "wings" of the primary candidate, close to the lines $\dot{r} = \pm 2\Delta r$ that pass through the primary candidate. For these candidates, the straight Nelder–Mead location refinement tends to converge on some suboptimal maximum in these wings; this point is generally close to the initial candidate found in the CG stage. By contrast, the new pattern search explores a much larger region of the r-$\dot{r}$ space; it will thus usually follow the wings towards the primary candidate, and then localise in on this

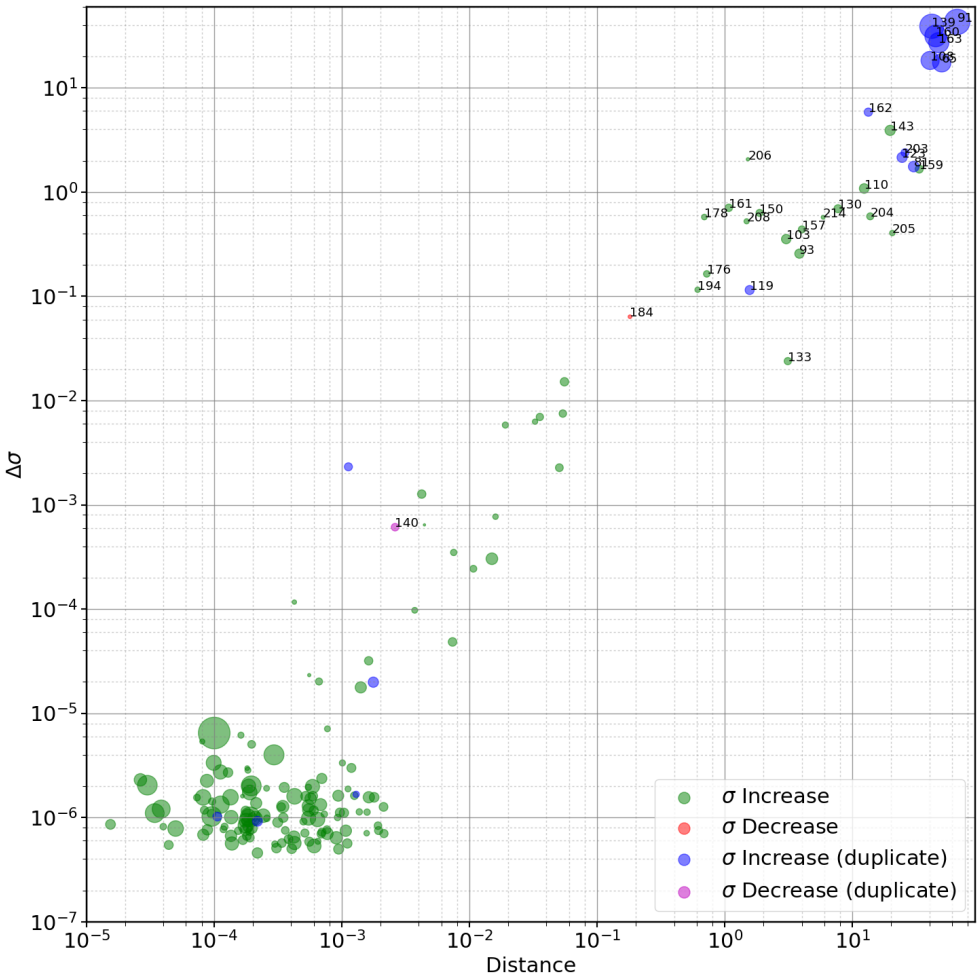


Figure 6.8: A comparison of the accuracy of the r-$\dot{r}$ location refinement components. Each circle represents an initial candidate. The horizontal axis is the euclidean distance between the maxima found by the old and new location refinement method; the vertical axis shows the difference in the sigma values of the corresponding maxima. The sign of this difference is indicated by the colour of the circles, with green and blue circles marking the cases where the new method found a better maximum – ie one with a higher summed power and thus a sigma value. The large cluster in the bottom left constitutes 50.45% of candidates, while 25.23% of candidates have a σ change of zero and thus cannot be shown in this figure. The new method is able to identify duplicates of other initial candidates. The cluster of duplicate candidates in the top right represents 3.15% of candidates, with the remainder making up 21.17%. The new method can be seen to produce a better maximum in all but two cases. The index of a number of candidates is shown beside them, several of these are discussed in later figures.

candidate. For wings with significant values to be generated, requires a very strong candidate, thus the primary candidate is usually found as some other stronger initial candidate in the CG stage. Therefore, these smaller initial candidates were not correctly identified as duplicates in the thinning of the initial candidates. These non-identified duplicate candidates are shown as the blue and magenta circles in the figure.

This second class of results usually makes up a small percentage ($\leq 3\%$) of the total candidates. In the existing implementation, these spurious candidates will be included in the final candidates and are left to the human operator to identify as duplicates. The new method, however, can leverage the higher computational power of the GPU to correctly identify these as duplicates and correctly cull them, simplifying the later analysis.

As an example, Figure 6.9 shows the actual optimisation planes of such a candidate. The location of the maximum found by the straight Nelder–Mead method (the blue cross) is very close to the initial location (the magenta cross), while the pattern search incrementally moves up the feature in r-$\dot{r}$ space to find the location of the strong primary candidate approximately 200 bin lengths away (the green cross). The first three iterations here are all at the first level, while the last three iterations are each a single plane at levels two, three and four. This shows how the new search refines a feature at a set scale, and only then hones the finer location.

If the first grid of the pattern search is not large enough, it may not correctly follow features and, in a similar manner to the Nelder–Mead search, it may localise in on a suboptimal maximum. However, if the size of these planes is increased while keeping the resolution constant, the number of points to be evaluated increases quadratically and can thus immensely increase computation time. Therefore, the initial size of the planes was chosen so as to balance accuracy and performance.

The third class of candidates are those that fall between the first two groups. These have some significant change in location and power, and this change is not caused by some other initial candidate. Thus, these are the cases in which the two methods found maxima that may be considered different to some degree, and these are of the greatest interest. These candidates can be seen as the candidates stretching from the bottom left to the top right of Figure 6.8. It was found that these can comprise up to 15% of the initial candidates. For the vast majority of these cases, the new method finds a larger maximum, to the extent that there are virtually no cases where the old

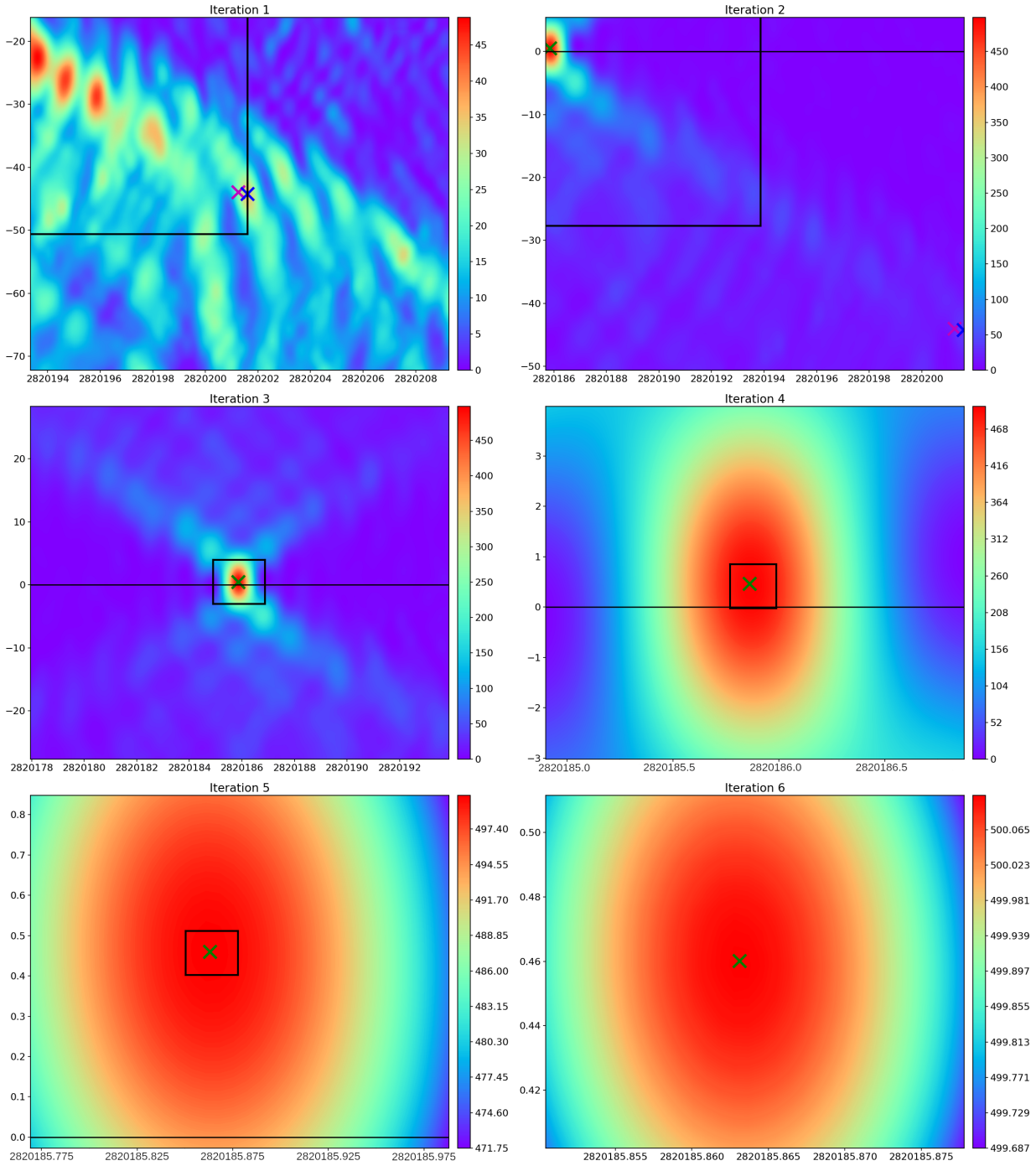


Figure 6.9: The iterative optimisation planes used by the GPU location refinement component to refine candidate 163 of Figure 6.8. The magenta cross marks the location of the initial candidate, the blue cross marks the maximum found by the old Nelder–Mead method, while the green cross marks the location of the maximum found by the new method. Here, the new method correctly follows the large-scale feature to find the true primary candidate, while the old method gets stuck at a suboptimal local maximum.

method found a candidate that had a significantly higher power. These candidates mark the most significant improvement of the new method over the old method.

The optimisation of two such candidates is shown here: one typical case (Figure 6.10), where the new method outperformed the old, and the one case in Figure 6.8 where the old method found a better candidate. In the first case (Figure 6.10) the Nelder–Mead optimisation simply climbed the slope it was on and settled on a suboptimal local maximum. however, using the new method, the first GPU iteration generated a plane significantly larger than the feature with a high enough resolution to reveal its basic structure; which has a greater maximum approximately two bin lengths away. The next iterations then zoomed in to refine the location of this higher maximum. This shows the typical and ideal case, in which there is only a single plane per level.

Figure 6.11 shows the one candidate in Figure 6.8 for which the old method gave a better result. This is a fairly weak candidate, and has quite a dispersed footprint in the r-$\dot{r}$ plane, with no obvious distinct maximum. As expected, the Nelder–Mead optimisation found a maximum very close to the initial location, whereas at the second level, the GPU optimisation selected a maximum on the right-hand side of the feature to refine. This maximum is slightly lower than the one found by the CPU. After investigation it was found that the cause of the misidentification was the accuracy in the second level (iteration 2). If the high-accuracy filters had been used at this level, the correct maximum would have been identified. It should be noted that the two maxima found are only 0.2 bin lengths apart, and may thus still be considered fairly similar.

By default, and in the cases discussed above, the double-precision refinement was done using the CPU Nelder–Mead optimisation. It is visible in the figures that after the four levels of single-precision refinement, the maximum is generally refined to a continuous dome-shaped surface. This type of feature is the ideal case to be optimised by the Nelder–Mead method.

It has been observed that a similar degree of accuracy can be achieved using the GPU alone by the addition of two levels to the template search. Each of the additional levels using high-accuracy filters, double-precision calculations, and a dimension as low as sixteen.

In conclusion, it is evident that the new location refinement almost always outperforms the existing method, often not only improving the location of the candidate, but correctly identifying duplicate candidates too. The latter is a subtle but significant improvement, since if these were

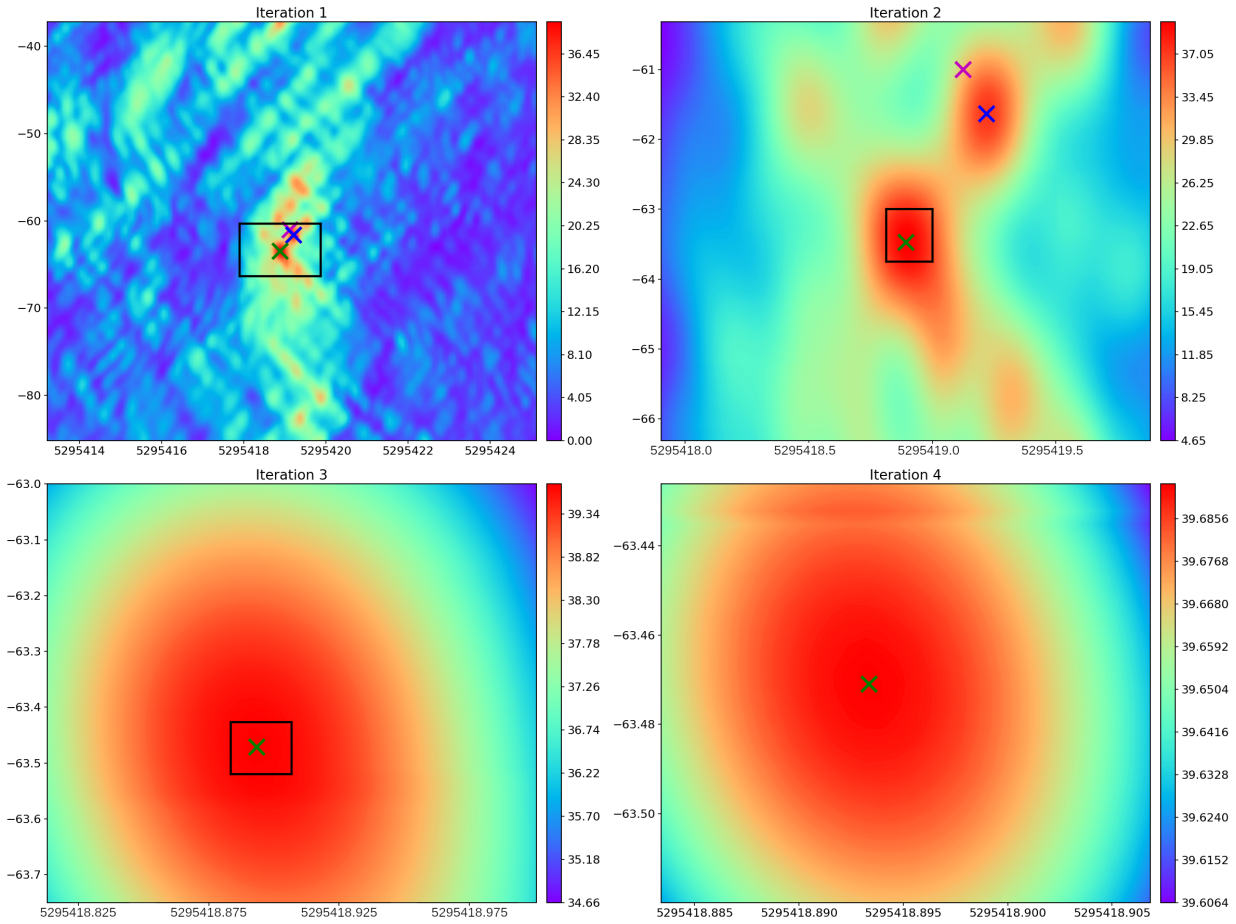


Figure 6.10: The iterative optimisation planes used by the GPU location refinement component to refine candidate 150 of Figure 6.8. The magenta cross marks the location of the initial candidate, the blue cross marks the maximum found by the old Nelder–Mead method, while the green cross marks the location of the maximum found by the new method. Here, the new method correctly identifies a better maximum for the candidate and refines this location, while the old method gets stuck at a suboptimal local maximum.

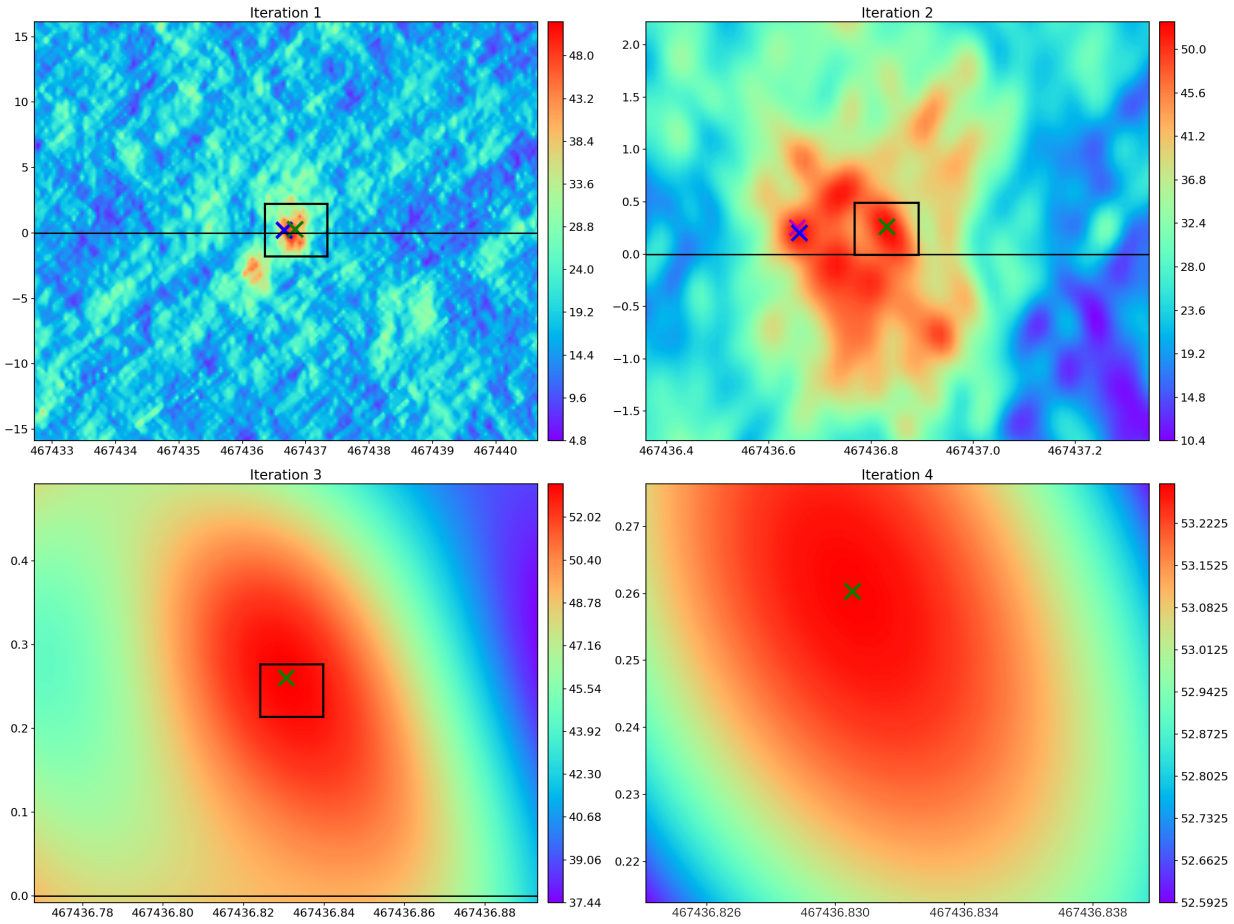


Figure 6.11: The iterative optimisation planes used by the GPU location refinement component to refine candidate 184 of Figure 6.8. The magenta cross marks the location of the initial candidate, the blue cross marks the maximum found by the old Nelder–Mead method, while the green cross marks the location of the maximum found by the new method. Here, the new method identifies a worse maximum which was refined, while the old method identifies and optimises a better maximum.

left for a human to identify, the additional time required would be comparatively vast. With the speed-ups achieved in this work, an entire search may be performed in a fraction of the time it takes a human operator to evaluate a single candidate, highlighting the need for automated candidate classification in large-scale searches.

Chapter 7

Results of the GPU acceleration

In this chapter, the performance of our GPU-accelerated FDAS is quantitatively evaluated and compared to the existing serial CPU implementation, as well as to the GPU implementation concurrently developed by Dimoudi et al.. The methodology and structure of our parallel computation is analysed, with a focus on how this was used to effectively reduce the run time of the FDAS.

The chapter begins with a methods section which describes how the timing was performed and what data was used. The section ends with a brief description of the main parameters of the search and their expected effects on its performance.

The timing analysis is separated into three sections: basic timing (Section 7.2), profiling (Section 7.3), and configuration (Section 7.4). The basic timing analysis examines the speed and speed-up of the search, which is run in the standard, fully asynchronous mode. In this first timing section, only the main search parameters are varied while the configuration parameters are left at their defaults or chosen automatically by the application; this is intended to give an indication of the expected "out-of-the-box" performance. When the search is run asynchronously, it is not possible to accurately time individual components, thus the lowest level of timing analysis attainable in this first timing section is the sub-stage level. This section concludes with a comparison of the performance of `AccelGPU` and the CPU-based `Accelsearch`, as well as the GPU-accelerated `AstroAccelerate`.

The second section – profiling analysis – is an examination of how the run time of the individual components scales with the search parameters. The search was run synchronously for these tests, to enable timing of the components.

The configuration analysis examines the individual effects of a variety of configuration parameters on a component level. For the most part, this required the search to be run synchronously to

allow accurate timing of the individual components. It is important to note, however, that a configuration which minimises the synchronous execution time of a specific component may not decrease the overall run time of the asynchronous search. For instance, a particular configuration may increase the occupancy of a specific kernel, decreasing its synchronous run time; however, when run asynchronously this may limit the number of other kernels that can run concurrently on the GPU, decreasing the overall run time. The number of configuration combinations is so high that examining them all in depth in asynchronous execution is not practical. It was found that, in most cases, the configurations that minimised the synchronous run time of a specific component, decreased the overall duration of the asynchronous search.

7.1 Methods

This section describes the methods by which the results reported here were obtained. Our implementation is integrated into the existing `Accelsearch` binary in such a way that GPU and/or CPU operation can be selected by command-line parameter. In typical use, one or the other would be selected; however, if both a CPU and GPU search are explicitly selected, the two stages of each type of search are run consecutively. This allowed us to time and compare the two searches using identical search parameters and data sets, run as one binary. All the timing results reported in this chapter represent the average of ten runs of a search using a particular configuration.

7.1.1 Basic timing

The timing of the full search, stages and sub-stages was performed in host code, using the `gettimeofday` function. The timing points were all located after a blocking GPU synchronisation, at a point where the application was running only a single CPU thread, giving an accurate wall time of the stages for both the CPU and the GPU search. For the analysis in this chapter, the run time of the full search is taken as the sum of the run times of the two stages. This was done to explicitly exclude the time required to: read the input data from file into main memory, initialise the CUDA context, and write the results to disc.

The CPU and GPU implementations have a number of subtle differences in their implementation, complicating direct comparison. The biggest of these differences is that in `Accelsearch`, the segment size is hardcoded, while we use a dynamic segment size (Section 5.2). To allow a fair

comparison between the two, for every test conducted, `Accelsearch` was recompiled with the same segment size used by the equivalent GPU search. This segment size should be close to optimal for both the CPU and GPU searches for the specific combination of $Z_{\max}$ and ω; and should give the best run time to allow fair comparison.

7.1.2 Profiling

In the profiling analysis, the run time of the individual components of the GPU search are examined. The various components run on the host were timed with the `gettimeofday` function, while the CUDA kernel executions and memory copies were timed using CUDA events and timers.

During a single search, most of the components are run many hundreds or thousands of times. For the profiling in this chapter, the run time of a component is taken as the sum of the run times of all occurrences of that component during a specific search. This allows the comparison of the same component across search configurations in which the specific component is run a different number of times, such as when running a search at two differing plane widths.

During asynchronous execution, the timing of CUDA kernels and memory copies may not be accurate when using events in separate, concurrent streams. The event marking the beginning of a CUDA kernel execution can occur in a stream well before the actual execution of the CUDA kernel begins. Other CUDA kernels running on the GPU may delay the launch of the kernel but will not delay the event. Even if accurate timing was possible, the timing results may not be relevant since they would be influenced by the concurrent execution of other kernels on the device. Thus, during profiling, the GPU search is run in synchronous mode (Section 5.3.1.2).

Timing using events and the `cudaEventElapsedTime` function has a resolution of $0.5\,\mu s$ [92, §4.5]. However, we have observed that consecutive queued kernel launches can be separated by as much as $30\,\mu s$. A single CUDA event marks the boundary between such kernels, thus the usable timing resolution of a single kernel may be in the tens of microseconds and will usually be inflated. This slight timing inaccuracy is amplified if many small instances are summed, such as when the run time reported is the sum of all instances of specific components.

7.1.3 Metrics

In many of the analyses to follow, neither the actual run time nor the speed per unit of time will be reported. Rather, the speed metric used is *search rate*, which is the number of a specific item that can be performed in the duration of the observation. This metric is used for a number of reasons. Firstly, a rate allows an easily comparable representation over a wide range of actual run times and observation lengths. Secondly, with the drive for real-time searching, this rate gives an indication of the resources required to undertake a search in real time. For instance, in a search covering 1000 DM trials, an FDAS search rate of 1000 would mean that the FDAS component of the overall pulsar search pipeline could be performed in real time, on the specific hardware used. However, a search rate of 500 would mean that twice the amount of the limiting hardware – GPUs, in this case – would be needed to perform the search in real time. This rate is specific to the task in question and excludes other computationally intensive tasks, such as dedispersion and FFTing the observation. In a large-scale survey, these other tasks can be run concurrently for differing DM trials and will have their own separate resource requirements. It is important to note that this metric is, however, directly linked to the sampling rate of the observation. For a set observation length, the number of frequency bins scales with the sampling rate. By default, `Accelsearch` will search frequency bins up to a maximum frequency of $10\,\mathrm{kHz}$. Thus, for sampling rates below $10\,\mathrm{kHz}$, the number of frequency bins searched will scale with the sampling rate; this means that the search rate will be inversely proportional to the sampling rate for values below this threshold.

7.1.4 Data

In this chapter, we aim to give timing results that are representative of real-world conditions. The content of data that is searched can have some impact on the run time, that is to say that the number and type of initial candidates found can increase the run time of a number of the components. The vast majority of the components of the CG stage are not affected by the content of the data – the two exceptions being the SAS and the candidate storage components. The more significant is the latter, which, as discussed in Section 6.2.3.7, can block GPU computation. By contrast, the run time of the CO stage is highly dependent on the content of the search, with the number of initial candidates directly scaling the run time of the CO stage.

Periodic RFI can cause a large number of spurious initial candidates and is expected to be the main cause of "excessive" initial candidates. Thus, it is desirable to perform timing with data that contains a fair amount of RFI, as this will give a fair representation of real searches.

For testing, a long observation containing a moderate amount of periodic RFI was selected. The data was a ∼7.7 hour observation of Terzan 5 taken with the Green Bank Telescope (GBT) in July of 2012. The observation was dedispersed at a DM of ∼239.93 pc/cm^3 and contains ∼6.8e8 samples recorded at a temporal resolution of 4.096e-5 seconds. This data was cleaned of prominent RFI using the `rfifind` tool in PRESTO, as per the PRESTO tutorial [113]. All the data sets used in this chapter are subsets of this observation. These time series were FFTed using the `realfft` binary from PRESTO and the resulting FTs were used as the input data for the various tests. The number of initial and final candidates for a search with a $Z_{\max}$ of 200 and summing 16 harmonics, are shown in Table 7.1. Despite the RFI removal, searches of this data still produce a large number of RFI-related candidates, which is specifically why this data was selected.

The sampling frequency of this time series is ∼24.414 KHz. By default, `Accelsearch` only searches the frequency range where the primary plane ranges from 1 Hz to 10 KHz, thus not the entire input DFT is searched. The duration, number of samples, and number of DFT bins searched for each observation are given in Table 7.1. It is noteworthy that the sampling rate of this data set is higher than is generally the case in pulsar searching, thus where search rates are reported, these values may be lower than expected with other searches.

7.1.4.1 SKA Data

This work is not intended to be used in the upcoming SKA pulsar searches, however, there is value in discussing how `AccelGPU` would fare when searching SKA data. The proposed SKA1-mid pulsar search observations are expected to have a sampling rate of ∼64 µs and contain 2^{23} samples, and will thus have a duration of 536.87 seconds [54, 66]. The proposed acceleration search of these data sets will cover 100 acceleration trials and have at least 500 DM trials [66].

These SKA parameters can be translated to our test data sets to give a rough equivalent. The frequency of the last DFT bin of the SKA data would only be 7.8 KHz, meaning that all the ∼4.2e6 DFT bins would be searched. If the number of DFT bins searched is translated to our test data sets, it would be equivalent to an observation length of 10.24e6 samples. One of our

Table 7.1: Details of the data sets used in testing. All these data sets are subsets of a single observation that contains a moderate amount of periodic RFI. The first column gives the number of time samples (N_t) in each data set; the second gives the number of DFT bins searched. The "initial candidates" column shows the number of candidate values found during the CG stage. The last column gives the actual number of candidates optimised, after culling harmonically and spatially related candidates.

Samples (10^6)	DFT Bins Searched (10^6)	Duration (Minutes)	Initial Candidates	Optimised Candidates
1	0.41	0.68	267	14
2	0.82	1.37	944	27
3	1.23	2.05	1127	37
5	2.05	3.41	1194	42
10	4.10	6.83	1444	78
15	6.14	10.24	1669	105
20	8.19	13.65	2351	125
25	10.24	17.07	1559	114
30	12.29	20.48	1767	132
35	14.33	23.89	1944	134
40	16.38	27.31	2399	156
45	18.43	30.72	2915	156
50	20.48	34.13	3144	188

standard observation lengths is 10e6 samples. This data set can thus be thought of as similar, in terms of the number of DFT bins searched, to the proposed SKA pulsar search data sets. Thus, the absolute search time of this data set can be used as a proxy for the SKA data. It is noted, however, that our data sets are sampled at a higher sampling rate, thus the duration of our 10.24e6 sample observation is 419.43 seconds – approximately 0.78 times the duration of the SKA observations. Thus, where search rates are reported, the values for the 10e6 sample observation can be inflated by 1.28 to give the equivalent rate for the proposed SKA search data sets. The search rate metric will automatically adjust for the minor difference in number of DFT bins searched between the 10e6 sample and the SKA observations. The 100 acceleration trials of the SKA search equate to a $Z_{\max}$ of 100 with the standard two Z resolution used by `Accelsearch`.

7.1.5 Parameters

Here, a brief outline is given of the main parameters of the search and their expected effects. As discussed in Chapter 3, these parameters are separated into two groups. The first, search parameters, affect the structure of the search, are implementation agnostic, and are expected to influence the number of candidates found. Configuration parameters, the second group, are specific to the implementation and are expected to affect the speed and, to some extent, the accuracy of the search.

7.1.5.1 Search parameters

These parameters change the range and resolution of the pulsar-specific parameter space searched.

Observation length

Observation length is one of the main parameters used to show the basic scaling of the search, and is expressed in number of samples (N_t). The run time of the search is expected to scale linearly with observation length, for the reasons that follow. In the CG stage, the input DFT is iterated through in batches; the processing of each batch is roughly equivalent with respect to computation. Increasing N_t will therefore increase the number of batches processed. This results in the run time of the CG stage scaling linearly with observation length. The run time of the CO stage is dependent on the number and properties of the actual initial candidates found, which is in turn dependent on many factors. However, one may expect some increase in the number of candidates found as observation length is increased. The reader is reminded that by default, `Accelsearch` searches the data at a frequency resolution of 0.5 bins (Section 3.3.1), thus the actual number of r values searched is double the range of the Fourier bins covered. These factors all contribute to the run time of the search scaling linearly with N_t.

$Z_{\max}$

The second scaling parameter is the number of acceleration trials to be searched. This number is set with the $Z_{\max}$ parameter, with searches covering the acceleration values from -$Z_{\max}$ to +$Z_{\max}$ at a resolution of 2 Z. Thus, $Z_{\max}$ relates directly to the height of the various planes of a batch and is expected to scale the run time roughly linearly. It is noted that in this work, a single resolution

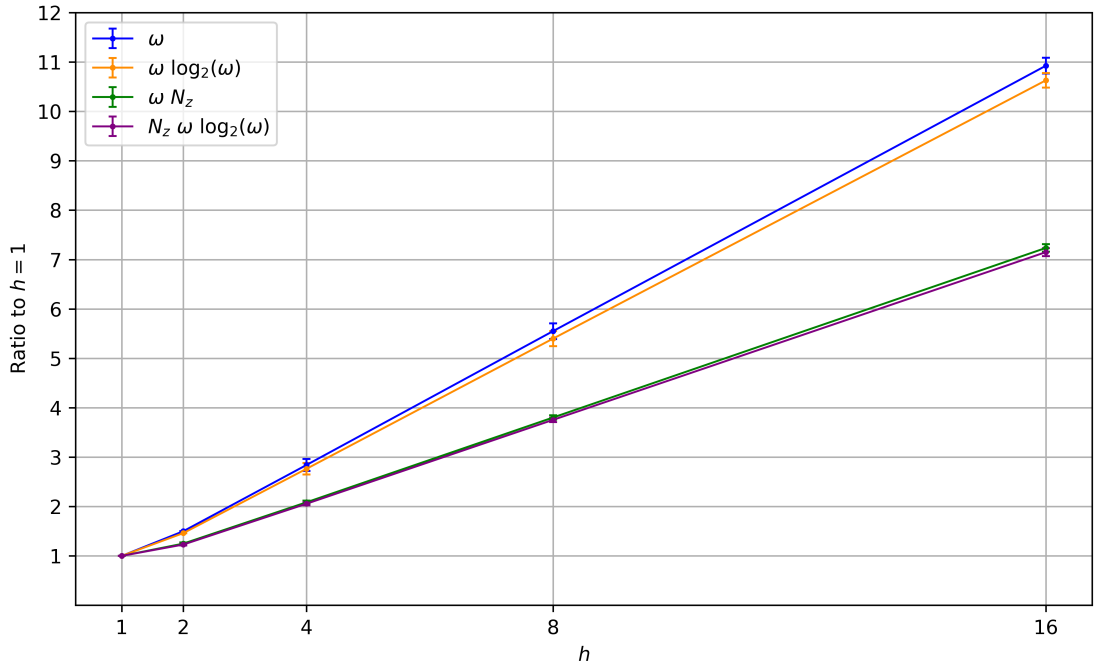


Figure 7.1: The scaling of a family of planes relative to a similar family containing only the primary plane. The points are the mean ratios of 901 families with ω ranging from 1 to 32, $Z_{\max}$ ranging from 0 to 400, and where the contamination in the primary plane is no more than 20%. The error bars represent the standard deviations of these 901 ratios. The top two lines can be used to predict how the run time of the components dealing with the input will scale with h. Similarly, the lower two lines can be used as a prediction of how the run time of the components dealing with the full planes will scale with h.

of 2 Z is used for all results reported. This resolution is hardcoded in `Accelsearch`, while it is run-time configurable in `AccelGPU`.

Harmonics

The last scaling parameter is h. Harmonic summing is performed in stages (Section 3.4.3); each additional stage doubles the maximum number of harmonics summed and includes the results of all previous stages. Thus, the h parameter is a power of two in the range: 1, 2, 4, 8 and 16.

The computational complexity of the key components and the number of points in a family of planes can be used to predict how run time will scale with h. The relative number of points in the various planes of a family is somewhat dependent on the ω and $Z_{\max}$ parameters, as the width of the various sub-planes and the decomposition of the sub-planes into stacks is influenced by these two parameters. This influence is small, however, and the relative number of points has only a

minor variation across the applicable range of these parameters. As an example, when summing two harmonics, the second plane will be placed in the second stack (Section 6.2.1). Thus, the width and height will be half that of the primary plane, meaning that the smaller stack will contain a quarter of the r-$\dot{r}$ points of the larger stack. A family of two harmonics will therefore contain 1.25 as many r-$\dot{r}$ points as a similar family containing only a single harmonic. Figure 7.1 shows this and similar values for the range of h. The sum of the relative plane widths is shown by the blue line, this can be used to predict how the run time of the components dealing with the input will scale with h. However, the FFT of the input data will not follow this scaling, rather it scales as $\mathcal{O}\left(\omega \log_2(\omega)\right)$, which is shown by the orange line. This scaling is very similar to the blue line, meaning that it can be used to estimate the scaling of the run time of the components dealing with the segments of the input DFT. The green line shows how the number of r-$\dot{r}$ points in a family of planes scales with h, this can be used to predict the scaling of the multiplication and SAS components. As before, the scaling of the iFFT component – shown by the purple line – differs only slightly from the green line. The scaling predictions above only hold for a standard search, as in an in-memory search, only the run time of the SAS component will scale with h. The SAS component is not easily isolated, so no analytical prediction is made for the in-memory search.

Apart from the transition from one to two harmonics, there is a roughly linear relationship between h and all components shown in Figure 7.1. One may expect the run time of the CG stage of the standard search to scale similarly to the green line Figure 7.1, assuming the run time is dominated by the components dealing with the full planes. It is noted that our prediction is similar to that made in the PRESTO tutorial: "The time that the searches take doubles for each additional level of harmonic summing" [113]. However, we note that this does not hold for the transition from one to two harmonics, and that a more accurate value for the scaling is an increase of 1.8 times for each additional stage of harmonic summing.

7.1.5.2 Configuration parameters

Here, a number of the high-level configuration parameters are outlined. These affect a number of components, therefore an understanding of these parameters is required for the discussion of basic timing in the next section. The details of many of the lower-level configuration parameters are left until Section 7.4.

Plane width

The ω parameter controls the width of the primary plane and thus the width of the largest stack of the batches used in the CG stage. The plane width is a power of two, usually in the range: 2048, 4096, 8192 and 16384, however in this analysis we extend these to include 1024 and 32768. In `AccelGPU`, the segment size is determined from ω (Section 5.2), which is a command-line parameter.

Plane width is an important configuration parameter and the optimal plane width is influenced by a number of factors. The first of these is that wider planes will have less contamination relative to narrower planes, and thus will reduce wasted computation. Counter to this, the run times of the various components scale differently with plane width, so that larger widths can disproportionately increase the run time of some of the components, particularly those using Fourier transforms. In addition, the plane width has some effect on the granularity of the search; wider planes may allow more efficient use of a GPU and this has some influence on the run time of the GPU search. Furthermore, the optimal plane width is expected to be affected by the number of harmonics summed and the $Z_{\max}$ of the search.

Number of segments per CG plan

In `AccelGPU`, N_s segments of the input DFT can be aggregated together and processed as a batch; in this work $1 \leq /ns \leq 12$. Grouping segments allows many kernels to operate on larger sections of data which can increase device utilisation. Furthermore, this can amortise some computation and memory transactions, thus allowing a higher degree of efficiency. However, many of the CUDA kernels require more registers as N_s is increased; this can negatively affect performance, especially if register spilling occurs.

Number of CPU threads

The main loops of the CG and CO stages are run asynchronously and divided amongst N_g and N_o CPU threads respectively. Each of these threads uses a separate plan data structure to process a number of iterations. Using multiple CPU threads allows concurrent CPU computation, and running multiple plans simultaneously will increase the number of concurrent asynchronous CUDA kernels launched. These both have the possibility of increasing the utilisation of the rel-

evant processing unit, which can reduce the asynchronous run time. However, having too many plans running concurrently can increase contention for the limited GPU resources, which can reduce the speed of the asynchronous pipeline. Furthermore, the memory transfers to and from the device associated with running each plan may cause unexpected synchronisation bottlenecks (Section 6.2.4.3). Both of these factors may contribute to a deterioration of asynchronous run time if N_g and N_o are increased beyond some point. Another factor is the amount of memory used; each plan data structure requires some device memory. The size of this memory can range from tens to several hundreds of megabytes depending on the ω, $Z_{\max}$, N_s and h parameters. Therefore, the amount of GPU memory can limit the number of plans and thus CPU threads used in the main loops of the two stages.

7.2 Timing

In this section, the search rate of the GPU-accelerated search, as well as the speed-up of our GPU search over the CPU equivalent is presented. The lowest level of timing shown here is the sub-stage, as timing of individual components is not possible when the search is run in normal asynchronous execution.

Wherever possible, the CG and CO stages are presented separately. The performance of the CG stage is expected, for the most part, to be independent of the content of the observation searched, with only the candidate storage and the SAS components being influenced by the number of initial candidates found in a specific observation. Thus, the timing results of the CG stage can be extended and extrapolated to structurally similar observations. By contrast, the run time of the CO stage is highly dependent on the number and properties of the actual candidates being optimised and can differ drastically between structurally similar observations. For observation with low RFI, fewer initial candidates are expected, and the CO time will be relatively low, to the point that it may be insignificant. However, as the number of initial candidates increase, the CO can dominate the run time of the full search. Thus, both stages are considered significant and are generally examined independently. The results of the CG stage can be used as best case results of the full search, while the results of the full search reported here serve as a worst case scenario in which the data being searched contains a large amount of RFI and thus generates many candidates.

The focus of this section is on the effect of scaling the three main search parameters: observation length, $Z_{\max}$, and h. There are a vast number of possible combinations of these three parameters, so reporting even a limited range of all combinations is not feasible. Consequentially, each is examined independently, scaling only the parameter of interest while keeping the other two constant. For the purposes of consistency, each parameter in this analysis has a set default value. The observation length chosen as the default value is the 10e6 sample SKA equivalent. A $Z_{\max}$ of 100 was used as the constant, since beyond this point, the change in acceleration starts to become nonlinear, decreasing the effectiveness of the correlation method. The value for h is set at the maximum of 16, as summing higher harmonics makes the search more sensitive to pulsars with narrow duty cycles. These defaults define what will be referred to as the *archetypal* search parameters.

The actual search rates reported are of some interest, however, they are linked to the structure and content of the observations as well as to the specific hardware used. The focus and, more significantly, the value of this analysis is how the search rate scales with the various search parameters. The in-memory and standard searches will be analysed side-by-side, as the choice of search is reliant on the memory available on a specific GPU.

For the most part, in this section, the various configuration parameters were left at their defaults, or were left for the application to determine. The majority of these parameters affect only the GPU search. As previously mentioned, the one notable exception to this is plane width. This is a parameter of the GPU search, while the equivalent value is determined from the hardcoded S_w in the CPU search (Section 5.2). It is important to note that the CPU and GPU searches have differing optimal plane widths, and their respective optimal values are affected by the values of other parameters. To allow a fair comparison between the two searches, the following allowance has been made in this section only. For both the CPU and GPU searches, the best run time of the plane widths in the range 2048, 4096, 8192 and 16384 is used. In this manner, all analysis and comparison is done between the optimal CPU and GPU plane widths. For each combination of $Z_{\max}$ and h, the optimal S_w was calculated and the CPU implementation recompiled with the appropriate S_w. If this was not carried out, the run times of the CPU implementation would have a sawtooth appearance and in some cases would be almost double their optimal value.

7.2.1 Speed

The following sections will report the timing results of this work, and begin with the principle metric: speed. The scaling of the speed will be examined across the three main search parameters: N_t, $Z_{\max}$ and h.

7.2.1.1 Observation length

The search rate of the archetypal search over a range of N_t is shown in Figure 7.2. The three panels show the search rate of the full search, as well as that of the CG and CO stages. The search rate of the CG stage has large sections that are roughly constant. This constant speed shows the expected linear scaling of run time with N_t. The CG stage has some variability and has a gradual increase in search rate with N_t. The run time of this stage is strongly linked to the number of candidates optimised, which is in turn related to the observation length, as can be seen in Table 7.1. It should be noted that the vertical scales of the three panels differ, and that the search rate of the CO stage is initially well below the CG stage, but quickly increases, becoming significantly greater than that of the CG stage. The lower the search rate, the greater the proportionate run time, thus for shorter observations, the run time of the CO stage is greater than that of the CG stage, and therefore the former dominates the run time of the full search for shorter observations ($N_t \lesssim 10\text{e}6$). This reduces the search rate of the full search for shorter observations, while the flattening off for longer observations can be attributed to the run time being dominated by the CG stage.

The centre panel shows the search rate of the CG stage, with in-memory searches shown by solid lines and standard searches as dashed lines. The search rate of the standard variant of the CG stage is almost constant, with only a marked drop-off for very short observations, while the in-memory variant has a number of clear features. These features are the result of different automatic configurations, dependent on the amount of available device memory. Below $N_t = 19\text{e}6$, the full in-memory plane can be stored in device memory; beyond this point, the in-memory plane needs to be split and searched in two halves. In this case, using a split plane reduces the speed of the CG stage by 7 to 16 percent, with the greatest reductions occurring when summing fewer harmonics. We note that this marked decrease in speed when having to use a split plane is not seen in all combinations of search and configuration parameters, the cause of this drop in speed is discussed

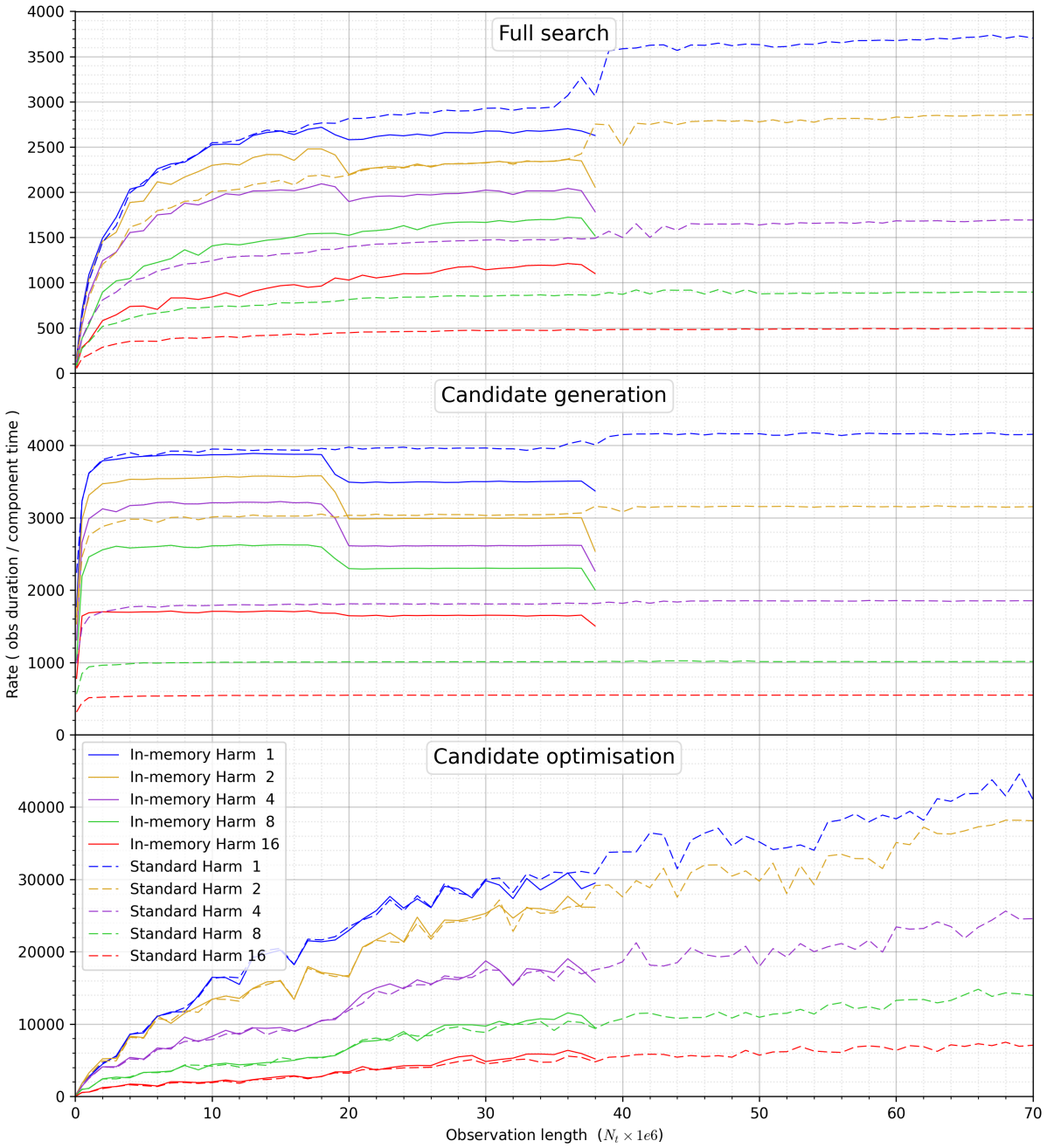


Figure 7.2: The speed of the search on a GTX 1070. The top panel shows the speed of the full search across a range of observation lengths. The centre and bottom panels show the speeds of the CG and CO stages. The standard variants of the search are shown by the dashed lines and the in-memory variants by solid lines.

in detail in 7.3.2.2. Beyond $N_t = 37\text{e}6$, there is insufficient device memory to store half the in-memory plane, meaning that beyond this point, an in-memory search cannot be performed. There are clear drop-offs in speed just before both of these points. As the size of the in-memory plane increases, there is less device memory available for the CG plan work areas, meaning that N_g and N_s need to be reduced, causing the decrease in speed; this is discussed further in Section 7.4.3.

In most cases, the in-memory variant of the search is the fastest. When summing only one harmonic, the two variants are a similar speed; when summing more harmonics, the in-memory variant is faster with the improvements increasing with the number of harmonics summed. Splitting the in-memory plane can have a negative impact on the search speed and should only be done when needed.

Figure 7.2 can be used to predict the speed at which one could search a single DM trial of a single beam of the proposed SKA1-mid data set. This data set is well within the range where an in-memory search can be performed. When summing 16 harmonics, the search rates for the CG stage, CO stage, and the full search are 1733.4, 2110.9 and 872.5 respectively. These rates can be inflated by 1.28 as the test data set is recorded at a higher sampling rate than the SKA data, meaning that the search rates of the SKA data would be 2218.8, 2701.9 and 1116.7 for the respective elements of the search. These results were obtained with an NVIDIA GTX 1070 – a desktop GPU from the Pascal generation. The test data used to generate these results contains a large amount of RFI and thus produces a large number of candidates; the majority of the SKA search data may be expected to contain significantly fewer candidates. Thus, the typical search rate of the full search is expected to be closer to the rate of the CG stage, although the run time of some data sets may approach those of our test data. To search 500 DM trials across 1500 beams in real time would require between 340 and 675 of these desktop GPUs. This reduces the computation requirements to a point at which the FDAS can be run in real time on a cluster of 500 compute nodes similar to the one proposed for use in the actual SKA survey [66]. The speed-up obtained could be even more pronounced given that this cluster is likely to use significantly more powerful compute-specific GPUs.

7.2.1.2 $Z_{\max}$

The scaling of run time with $Z_{\max}$ is slightly more complex than with observation length. There are two main factors that influence the scaling of the CG stage with $Z_{\max}$. The first is that for a single batch, the components that deal directly with the input are unaffected by $Z_{\max}$, while the multiplication, iFFT and SAS components are expected to scale linearly with $Z_{\max}$. If one assumes that the run time of the CG stage is dominated by the latter three components, one would expect it to scale roughly linearly with $Z_{\max}$. The second factor is that the $Z_{\max}$ influences the amount of contamination, and thus the overlap between consecutive segments (Section 5.3.3.1). The effect of this contamination is discussed in Section 7.4.1, but for the moment it is assumed that it is insignificant.

To investigate the effect of scaling $Z_{\max}$, the metric Z-speed is used, this is the number of acceleration trials searched per second, for a given observation. Figure 7.3 shows the Z-speed for the archetypal search over a range of $Z_{\max}$. As with the previous section, the Z-speed of the full search is shown in the top panel, with the CG and CO stages below that.

There are a number of similarities between this figure and Figure 7.2. In the in-memory variant of the CG stage, the clear feature at a $Z_{\max}$ of $\sim$190 is the point at which the in-memory plane needs to be split. Similar to observation length, there is a noticeable drop-off in speed before the split and towards the end of the in-memory domain, caused by the decrease in available device memory. By contrast to the data shown in Figure 7.2, splitting the in-memory plane does not cause a marked drop in speed. The flat sections in the CG stage show the expected, roughly linear scaling of run time with $Z_{\max}$. Unlike observation length, it is evident that for small values of $Z_{\max}$, there is a steep drop-off. This drop-off can be attributed to the CG stage having some "constant" non-zero run time for a $Z_{\max}$ of 0. This is a result of the input normalisation and FFT components which do not scale with $Z_{\max}$. It can be observed that the Z-speed of the CG stage is not truly flat; instead, it appears to drop off for very large $Z_{\max}$. This drop-off is a result of the increased overlap between segments as $Z_{\max}$ is increased (Section 7.4.1).

It is apparent that the Z-speed for the CO stage is close to linear, scales with $Z_{\max}$ and has a zero intercept. This indicates a close-to-constant run time. This is due to the number of candidates found not scaling strongly with $Z_{\max}$, as the majority of the RFI-related candidates have

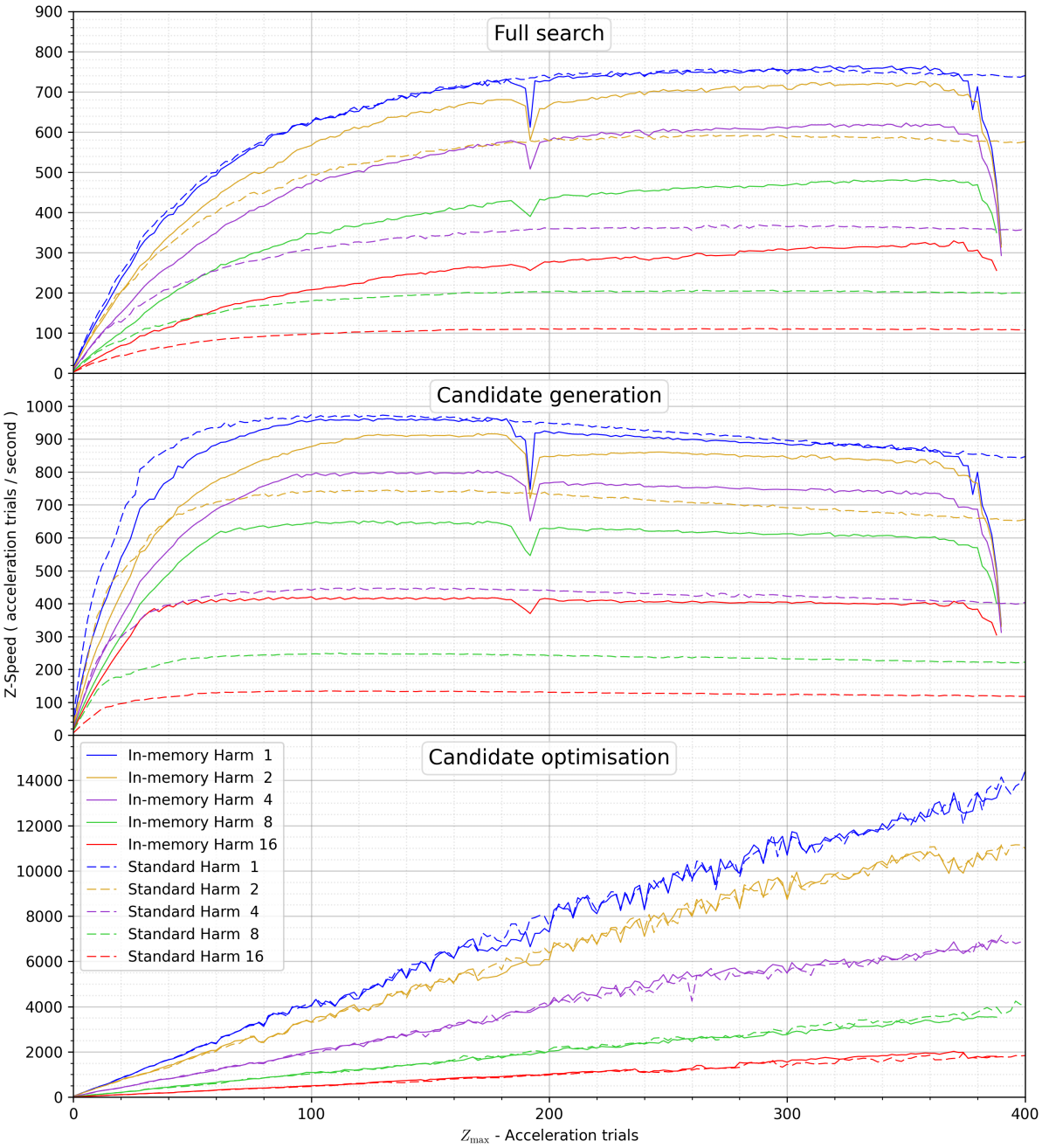


Figure 7.3: The speed of the search on a GTX 1070. The top panel shows the speed of the full search across a range of $Z_{\max}$. The centre and bottom panels show the speeds of the CG and CO stages. The standard variants of the search are shown by the dashed lines and the in-memory variants by solid lines.

accelerations close to zero; thus, increasing $Z_{\max}$ does not produce many more candidates.

Comparison to direct convolution

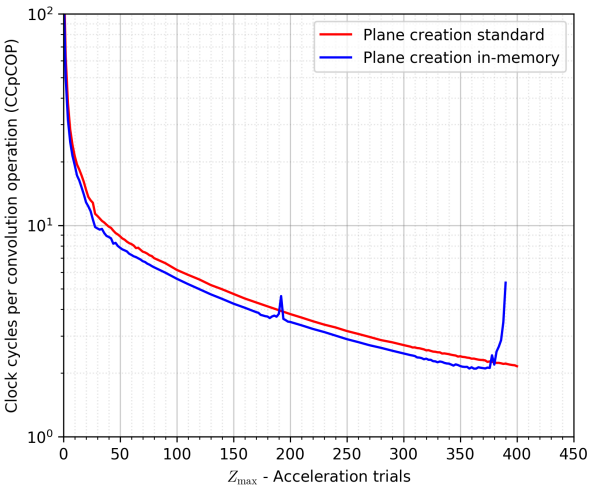


Figure 7.4: The speed, measured in CCpCOP, at which r-$\dot{r}$ points are generated in the CG stage of a standard search with $h = 0$ and $N_t = 10\text{e}10$, run on the GTX 1070.

Using fast convolution to generate large-scale r-$\dot{r}$ planes is usually much faster than direct application of the convolution. In Section 6.1.2 it was shown that large r-$\dot{r}$ planes can be generated at a speed of ∼25 CCpCOP using direct convolution. The speed at which r-$\dot{r}$ points are generated in the CG stage can be calculated, in CCpCOPs, using the theoretical number of COPs needed to generate all the r-$\dot{r}$ points and the run time. This speed scales with $Z_{\max}$ (Figure 7.4) to a maximum of 2.1 CCpCOP[1], which is significantly better than the ∼25 CCpCOP obtained when performing direct convolution (Section 6.1.2). This direct comparison of the two methods is not a "fair" one, as the direct convolution calculates all the filter coefficients on the fly, which is unnecessary when calculating the fixed resolution r-$\dot{r}$ planes of the CG stage. Nonetheless, the direct method could never attain a speed as high as two CCpCOPs, since at minimum, direct application of a COP consists of a complex multiplication and addition, which will always take more than two clock cycles. This clearly shows the superiority of the fast convolution method for generating large fixed resolution r-$\dot{r}$ planes.

7.2.1.3 Harmonics summed

In Section 7.1.5.1, the number of r-$\dot{r}$ points in a family of planes was used to predict how the run time of the CG stage will scale with h. Table 7.2 shows the observed mean ratios of the run times of the CG stage for a range of $Z_{\max}$ values from 50 to 400. The observed values are in excellent

[1]This is not two clock cycles per r-$\dot{r}$ point. At $Z_{\max} = 400$, an average of 240 COPs is required per r-$\dot{r}$ point, giving an average of 504 clock cycles per r-$\dot{r}$ point.

Table 7.2: Harmonic ratios. The run time of the CG stage relative to a search of one harmonic. The values are the mean for a range of searches with $50 \leq Z_{\max} \leq 400$, $N_t = 10e6$.

h	GTX 770 standard	GTX 970 standard	GTX 1070 standard	`Accelsearch` standard	GTX 770 in-memory	GTX 970 in-memory	GTX 1070 in-memory	`Accelsearch` in-memory
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.29	1.26	1.25	1.40	1.04	1.05	1.07	1.33
4	2.17	2.07	2.08	2.37	1.12	1.15	1.17	1.56
8	3.92	3.78	3.32	4.27	1.33	1.34	1.35	1.92
16	7.46	7.30	8.16	7.99	1.77	1.83	1.93	2.60

agreement with the ratios of r-$\dot{r}$ points. However, all the ratios are slightly above the predictions; this is due to a number of the minor components not having the same scaling as the dominant ones. The predictions of Section 7.1.5.1 do not hold for the in-memory variant of the search. The observed scaling of this variant is given in the second half of Table 7.2. As expected, the relative run times of the in-memory searches are far lower than those of the standard searches, with the summing of 16 harmonics taking approximately twice the time of that of searching a single harmonic. The same ratios are presented for both the standard and in-memory CPU searches.

Increasing the number of harmonics summed increases the sensitivity of a search to pulsars with low duty cycles. Thus, it is desirable to perform a number of stages of harmonic summing. The results above show that when performing an in-memory search, the cost of additional harmonic summing is comparably low. Thus, if it is possible to do an in-memory search, it would be advisable to sum a large number of harmonics. On the other hand, when a standard search must be used, the marginal cost of searching higher harmonics is greater, with a search of 16 harmonics taking 7 to 8 times longer than the search of a single harmonic. Therefore, when performing a standard search, the additional cost of summing higher harmonics must be taken into consideration.

7.2.1.4 In-memory search

From the previous results it is evident that the in-memory search is generally faster than the standard search. The speed-up of the CG stage of the in-memory variant over the standard variant is given in Table 7.3. The values for the CPU and GPU searches are similar; the notable exception is $h = 1$. For the GPU searches of one harmonic, the in-memory search can perform worse

Table 7.3: Speed-up of an in-memory search over a standard search. The values represent the mean speed-up when using an unsplit in-memory plane.

h	GTX 770	GTX 970	GTX 1070	`Accelsearch`
1	0.74	0.88	0.94	1.00
2	0.92	1.07	1.09	1.04
4	1.43	1.59	1.67	1.49
8	2.18	2.48	2.30	2.12
16	3.14	3.53	3.97	2.88

than the standard search on older GPUs. In Figures 7.2 and 7.3, it can be seen that the non-split in-memory search generally outperforms the standard search. For searches of 4, 8, and 16 harmonics, the in-memory search consistently outperforms the standard search, with a speed-up of 3–4 times for the search of 16 harmonics, with better speed-ups seen on the newer GPUs. `AccelGPU` will, where possible, automatically select an in-memory search when summing 2, 4, 8, or 16 harmonics and will do a standard search when summing one harmonic, if not specifically configured otherwise.

The CO stage is largely unaffected by the variant of search that is performed. However, it is important to note that the two variants do not generate identical initial candidates, although they are usually very similar. This slight difference is caused by the normalisation of the sections of input data. In both searches, each section of the input DFT is normalised using median normalisation; the lengths of data used differ in the two variants, thus the medians may differ as well. This causes the two methods to generate similar but different r-$\dot{r}$ values, for harmonically related planes.

7.2.2 Speed-up

In this section, the performance of our implementation, `AccelGPU`, is compared to the existing CPU implementation, `Accelsearch` as well as `AstroAccelerate` the GPU implementation under current development. This analysis compares the CPU in-memory search to the GPU in-memory search, and the standard CPU search to the standard GPU search. As in the previous section, the run times used are the best of a selection of plane widths, allowing a fair comparison between the CPU and the GPU implementations.

7.2.2.1 Observation length

Figure 7.5 shows the speed-up of both the in-memory and standard variants of the search, over a range of observation lengths, for a $Z_{\max}$ of 100. The speed-up of the standard search is generally greater than that of the in-memory search, with a drop-off in speed-up for shorter observations ($N_t \lesssim 4\text{e}6$). There is a close-to-constant speed-up in the CG stage, with the step from the single to the split in-memory plane clearly evident at $N_t \approx 19\text{e}6$. There are a number of significant "spikes" in the speed-up of the CG stage. These spikes are due to an increase in the run time of the existing CPU implementation; these abnormally long run times are specifically caused by the CPU candidate storage component. In these searches, the inefficiency of adding and removing many elements from the linked list used to hold the initial candidates causes a significant drop in the speed of those searches. This highlights the need for a more efficient method of storing initial candidates (Section 5.3.3.11).

The speed-up of the CO stage has more variability than that of the CG stage, and it is generally lower. This high variability is caused by the Nelder–Mead method used by `Accelsearch`, which is an iterative process; the number of iterations can vary significantly (Section 5.4.1), and are sensitive to the properties of the initial candidates. When observation length is short, the lower speed-up of the CO stage results in a drop-off in the speed-up of the full search. This is a result of the CO dominating the run time in this range. The speed-up of the full search increases with increasing observation length, as the run time of the CO stage becomes comparatively less significant. This means that the speed-up should be close to that of the CG stage for longer observations, or observations that find few initial candidates. As with the speeds, the speed-ups increase with h, with searches of 1 harmonic running 30 to 45 times faster, and searches of 16 harmonics running 60 to 70 times faster than the equivalent CPU search.

7.2.2.2 $Z_{\max}$

Figure 7.6 shows how the speed-up changes over a range of $Z_{\max}$ for the archetypal search. It can be seen that for the CG stage, the speed-ups start low (< 10) but rapidly increase with $Z_{\max}$, this continues to a $Z_{\max}$ of $\sim$50, whereafter they begin to flatten off. However, there is still a slight increase as $Z_{\max}$ increases, especially for the lower harmonics. These initial low speed-ups are a result of the decreased efficiency associated with processing the smaller planes that result

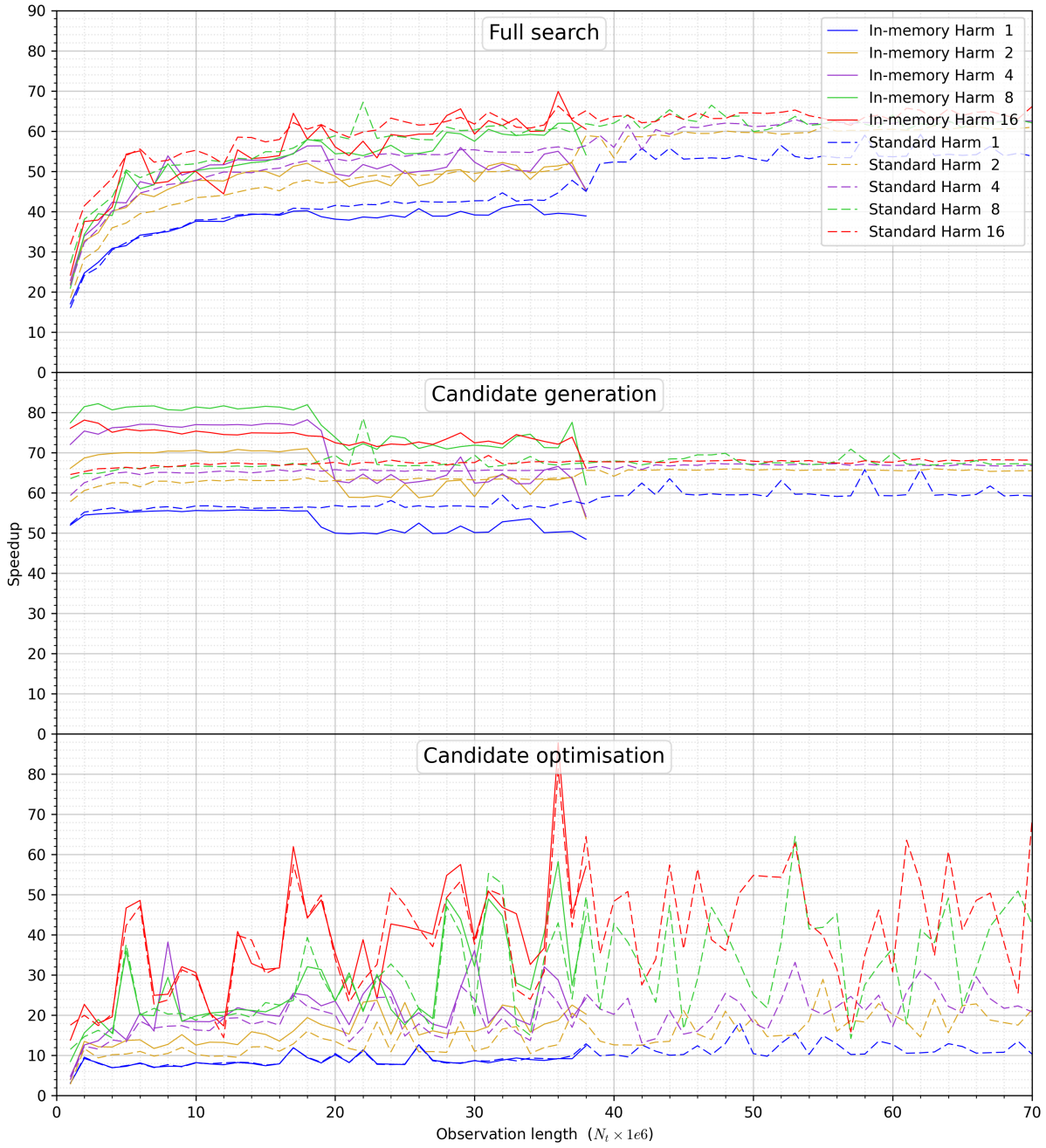


Figure 7.5: The speed-up of the `AccelGPU` run on a GTX 1070 over the serial CPU implementation. The top panel shows the speed-up of the full search across a range of observation lengths. The centre and bottom panels show the speed-ups of the CG and the CO stages. The standard variants of the search are shown by the dashed lines and the in-memory variants by solid lines.

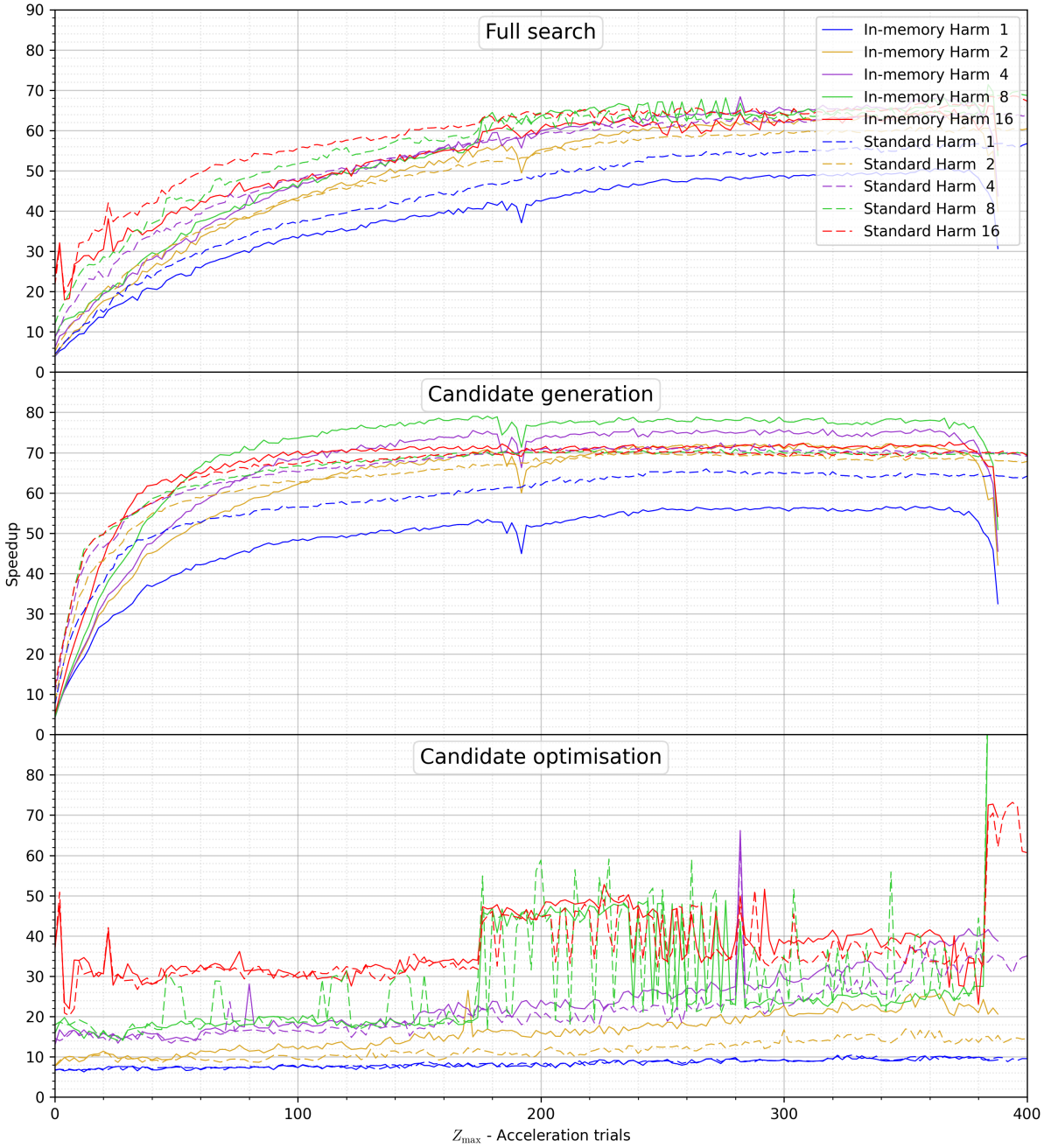


Figure 7.6: The speed-up of the archetypal search run on a GTX 1070 over the serial CPU implementation. The top panel shows the speed-up of the full search across a range of $Z_{\max}$. The centre and bottom panels show the speed-ups of the CG and the CO stages. The standard variants of the search are shown by the dashed lines and the in-memory variants by solid lines.

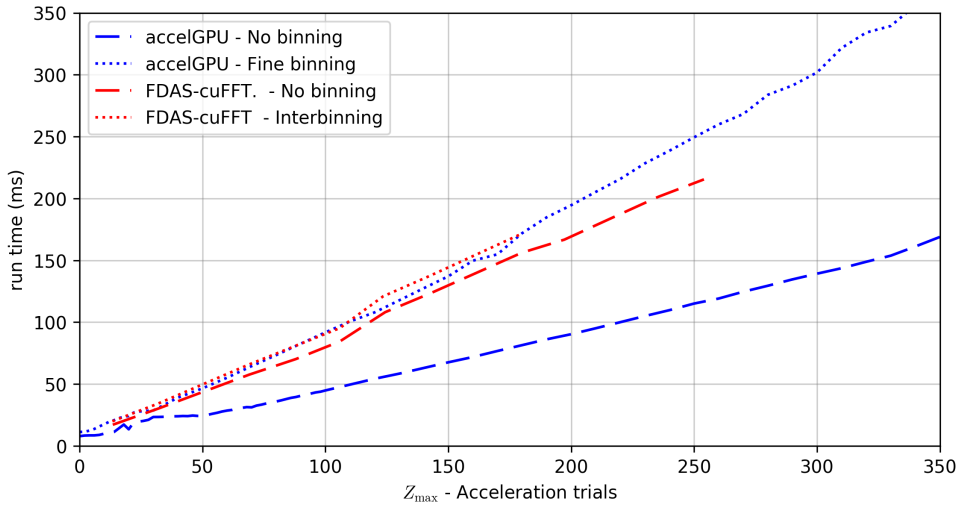
from lower $Z_{\max}$ values. For larger values of $Z_{\max}$, the speed-ups of the CG stage are in the 40–80 range, similar to those in the previous analysis. The severe dips in the CG stage of the in-memory search are due to the decreased efficiency of the GPU search, resulting from the decrease in available memory.

The speed-ups of the CO stage have sections that are fairly linear and in the range of 10–40 times. These are interspersed with sections that appear to oscillate between two levels. This oscillation is caused by the variation in the run time of the `Accelsearch` optimisation of a single candidate. In some cases, the Nelder–Mead method does not converge, meaning that many repetitions are performed, significantly increasing the run time of optimising a single candidate. Similar searches with slightly different $Z_{\max}$ values will cause the initial candidate in question to be found at a slightly different location, resulting in the Nelder–Mead method converging. As with the CG stage, the speed-up is greater as more harmonics are summed, and the workload is increased.

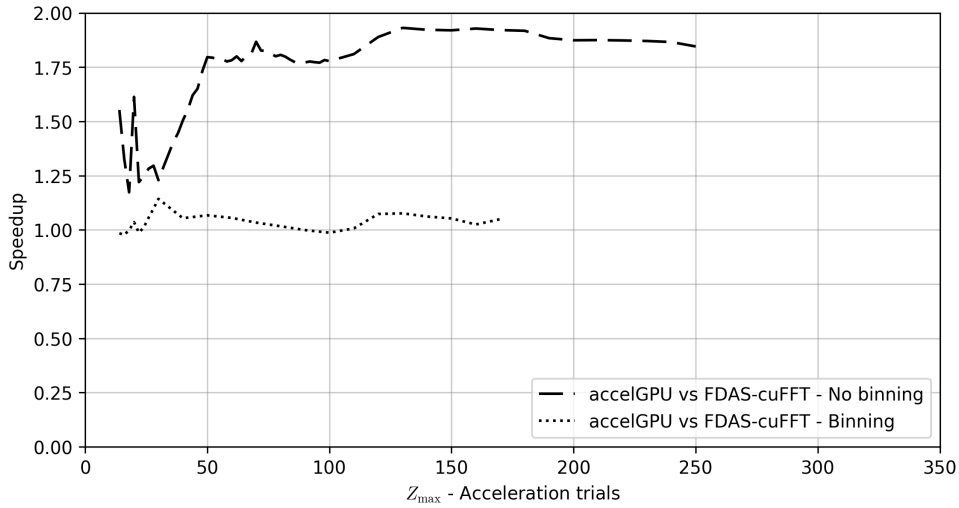
In Section 5.1, Amdahl's law was used to predict that the maximum speed-up attainable in the standard search would be 25.64 times, and 42.06 times for the in-memory search. These predictions were based on a search with $h = 16$ and $Z_{\max} = 200$. The actual speed-ups attained for these searches were 47 and 51 respectively for the standard and in-memory searches. Clearly, these speed-ups exceed the predictions, this can be attributed to the non-component computation not being truly serial in nature. Thus, much of the administration is spread across multiple CPU threads in the main loops of both the CG and CO stages, producing the better-than-predicted results.

7.2.2.3 Comparison to AstroAccelerate

Figure 7.7 shows a direct comparison of `AccelGPU` with one of the GPU-accelerated FDAS implementations created by Dimoudi et al., which were discussed in Section 4.6.2. Dimoudi et al. only include the plane creation task, of which they have two implementations: the first (FDAS-cuFFT) uses the cuFFT library [100] and the second uses a custom iFFT CUDA kernel incorporating the multiplication. This makes the latter significantly faster than the former (by 1.5–3.1 times). They use interbinning whereas `AccelGPU` uses fine binning, both increase the r resolution of the r-$\dot{r}$ plane generated; however, interbinning is less accurate but much faster.



(a) Run times of FDAS-cuFFT and `AccelGPU`



(b) Speed-up of `AccelGPU` vs FDAS-cuFFT

Figure 7.7: A comparison of `AccelGPU` and the FDAS-cuFFT implementation created by Dimoudi et al.. The FDAS-cuFFT run times were taken from Figure 6 in Dimoudi et al.. The run times of `AccelGPU` were obtained from the plane creation sub-stage of an in-memory search, run on a 2^{23} sample observation with a $64\,\mu s$ sampling rate. The two methods using no binning are very similar, however, our implementation runs significantly faster. When increasing the r resolution, Dimoudi et al. use interbinning, which is less accurate but substantially faster than the fine binning used by `AccelGPU`. Therefore, the performance of the two higher resolution searches are very similar.

The red lines in Figure 7.7a show the run times of the FDAS-cuFFT method, both with and without interbinning, run on an Nvidia M40. These timings are of plane creation only, and are thus comparable to our plane generation sub-stage (Section 5.3) of an in-memory search, with the same r resolution[2], run on similar data. The method used in `AccelGPU` to run a search with no binning is very similar to the FDAS-cuFFT, both use custom multiplication kernels and cuFFT to perform the iFFTs. For $Z_{\max} > 50$, `AccelGPU` is approximately 1.85 times faster than that of Dimoudi et al. (Figure 7.7). These results were obtained using a GTX 1070 which has 10% lower computational power and memory bandwidth than the M40. Comparing these two similar methods clearly shows the strength of our implementation, however Dimoudi et al. use two additional methods that make their best plane generation method significantly faster than ours. When using fine binning – as in `AccelGPU` – doubling the r resolution doubles the total work needed to generate the r-$\dot{r}$ plane, whereas interbinning is applied after the r-$\dot{r}$ plane has been created, making it significantly faster. Thus, when comparing the two searches with increased r resolutions, the two have very similar run times, as shown by the dotted line in Figure 7.7. Therefore, `AccelGPU` and the FDAS-cuFFT implementation of Dimoudi et al. have very similar run times, however `AccelGPU` uses a more accurate method of calculating the higher resolution r-$\dot{r}$ points. However, the custom iFFT method of Dimoudi et al. will generate r-$\dot{r}$ planes approximately twice as fast as `AccelGPU`. We note that implementing a custom iFFT into `AccelGPU` may significantly reduce the run time of the plane creation task.

7.3 Profiling

In this section, the run times and scaling of the key components of the GPU search are analysed. For the most part, the configuration parameters were left at their defaults, meaning that the values used should be similar to those of the asynchronous searches discussed in the previous section. One notable exception is that in synchronous execution, only one CG plan is used ($N_g = 1$); this reduces the amount of working memory needed on the device, which can affect the value of N_s selected by the application. For the moment, any effects this may have will be disregarded, as they are discussed in detail in Section 7.4.3.

In this section, we will show that the three most important components of the CG stage are

[2]In `AccelGPU`, the r resolution of the CG stage can be set with a run-time parameter allowing us to run the search with the same two resolutions as Dimoudi et al..

multiplication, iFFT and SAS, we refer to these as the "critical components". These components are all run on the GPU and have the longest run times. In most real-world use cases, the run time of the CG stage is almost completely determined by these components. Thus, much of the analysis and discussion that follows focuses on these components.

7.3.1 Components

Here, we discuss the performance of some of the components of the search. The run times of the components of the archetypal search are shown in Figure 7.8, which shows the components of both the standard and in-memory searches, on all three generations of GPUs. All the searches shown in Figure 7.8 have a plane width of 4096, which is not optimal for all the devices. However, this width allows a single unsplit in-memory plane to be used on all the GPUs tested, making the components as similar as possible across the devices.

7.3.1.1 CG initialisation

The CG initialisation components have been omitted from the figure, as their run times are low enough to be considered insignificant for a search of this length. The entire initialisation takes approximately $20\,\mathrm{ms}$, with the coefficient calculation and kernel FFT components taking only $450\,\mu\mathrm{s}$ and $310\,\mu\mathrm{s}$ respectively. Thus, the computational components of the initialisation take approximately two orders of magnitude less time than the full initialisation, with the vast majority of the initialisation time spent on allocating the host and device memory needed by the CG plans. This is in stark contrast to the CPU equivalent, where these two computational components account for over 97% of initialisation (Section 5.1).

7.3.1.2 Input

There are four components associated with processing the sections of the input DFT: input preparation, normalisation, input FFT, and an H2D memory transfer. Both the normalisation and the input FFT are computational components and can be performed by either the CPU or the GPU, the run times of both implementations are shown in Figure 7.8. The GPU implementations of both components for both the standard and in-memory searches are significantly faster than their CPU equivalents. As discussed in Section 5.3.3.3, median normalisation is not ideally suited to

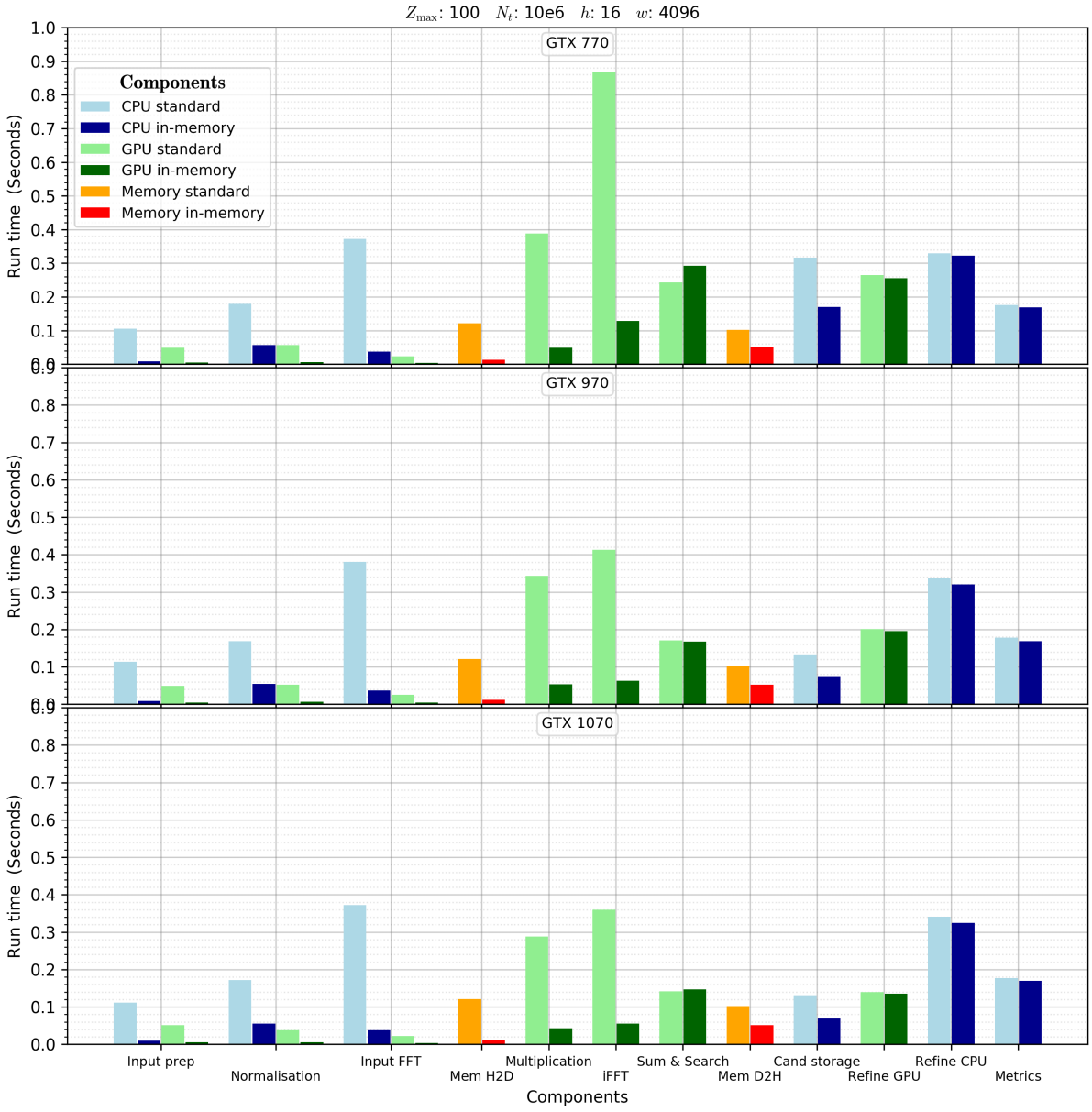


Figure 7.8: The run times of the key components of the archetypal search. The CPU components are shown in blue and the GPU components in green, with memory transactions in orange. The lighter and darker colours represent the in-memory and standard variants respectively.

GPU computation, thus with $\omega = 4096$, the GPU normalisation is only approximately four times faster than the CPU normalisation. The efficiency of the GPU normalisation, compared to the CPU normalisation, increases as the plane width increases. Thus, in a standard search, when h is decreased, there are fewer short sections of input data and the speed-up over the CPU implementation increases. This inverse relationship between speed-up and h differs to the standard behaviour, which was shown in Section 7.2.2. In addition, the normalisation in the in-memory search has a slightly better speed-up compared to the standard search equivalent, which uses only one wide section of input data for each segment, contrasting with the multiple shorter sections used in the standard variant. For the same reasons, if the plane width is increased to 16384, the speed-up of the normalisation in the standard search increases to approximately ten times.

The input FFT component has a more standard speed-up. With a plane width of 4096, the speed-up of the input FFT in a standard search is approximately 15, while this speed-up drops to approximately 7 with an in-memory search. These values rise to approximately 25 and 17 times, when the plane width is increased to 16384. In a standard search, the speed-up of the FFT component increases with h in a similar fashion to what was observed in Section 7.2.2.

The input preparation component (Section 5.3.3.2) is essentially the administration required to prepare the host-side input memory. This component is always performed by the CPU, however, Figure 7.8 shows two GPU run times for this component. In reality, these are all host-side run times, however the bars for the two different processing units are used to distinguish between the cases where the next component (normalisation), is run on the CPU or the GPU. When normalisation is performed on the CPU, the host-side memory is zeroed before normalisation begins, while on the GPU, the zeroing is performed as part of the normalisation CUDA kernel. Thus, the run time of the input preparation component is affected by which processing unit is used to perform the normalisation. It is evident that simply zeroing the host memory approximately doubles the run time of this component.

7.3.1.3 In-memory search

The typical differences between the components of the standard and in-memory searches can be seen in Figure 7.8. A number of components of the standard search take significantly longer than those of the in-memory search; these include the normalisation, input FFT, multiplication,

iFFT, and H2D memory copy components. With an in-memory search, only the primary plane is created, while with a standard search, all the sub-planes of a family of planes are additionally created. The components mentioned above are those associated with creating a section of r-$\dot{r}$ plane. Creating only the primary plane is equivalent to creating the plane for a search summing only a single harmonic. Thus, the in-memory run times of these components should be similar to their equivalents in a standard search of only one harmonic. It may therefore be expected that the ratios of the run times of these components may be close to the predictions made in Section 7.1.5.1.

For the archetypal parameters, the ratio of the number of input r-$\dot{r}$ points used for the in-memory and standard searches is $1 : 11.1$. The run times of the H2D memory copies differ by almost exactly this amount (Figure 7.8). However, the computational components dealing directly with the input data – normalisation and input FFT– do not show this predicted scaling. This difference may be explained by the computational complexity of these components, which is nonlinear, and indeed the method differs between the CPU and GPU implementations. In Figure 7.8, the difference in the multiplication and the iFFT components almost exactly matches the ratio of r-$\dot{r}$ points in the two families of planes ($1 : 7.3$); we find that this is generally the case for all stages of harmonic summing.

7.3.1.4 Sum-and-search

The run time of the SAS component should be similar between the in-memory and standard searches, as a similar number of operations and memory accesses are performed, only differing in the locations of the device memory accessed during the SAS component. This is the case on the GTX 1070 and the GTX 970, while on the GTX 770, the in-memory SAS component has a slightly longer run time than the equivalent standard search. The GTX 770 has approximately a third of the L2 cache of the GTX 970. The SAS kernels for the standard and in-memory searches are designed such that, in a single warp, each memory location of the powers plane will be read only once. The memory access patterns of both SAS kernels are designed in such a way that the threads from a single warp will simultaneously read a continuous block of memory ≤ 32 elements wide, resulting in a single coalesced memory transaction, making these memory reads very efficient. Many threads in separate warps and thread blocks will read spatially close memory

locations, especially when summing higher harmonics. These memory reads may benefit from L2 caching across warps and thread blocks. The more dispersed memory access pattern of the in-memory SAS kernel results in fewer L2 cache hits, in the smaller L2 cache of the GTX 770, resulting in the slightly longer run times on the older GTX 770.

7.3.2 Scaling

As in the case of the timing analysis (Section 7.2.1), the effect of scaling the three main search parameters – $Z_{\max}$, N_t, and h – are examined here to determine how they affect the run times of the individual components.

7.3.2.1 Observation length

The scaling of the run time of the components of the CG stage with N_t is fairly trivial. As previously noted, due to the iterative nature of the search, increasing N_t simply increases the number of batches processed in the CG stage. The processing of each batch is essentially the same, thus scaling N_t results in a simple linear scaling of the run times of all the components of the CG stage, as well as the asynchronous run time of the entire stage, as seen in Section 7.2.1.

The scaling of the CO stage with N_t is slightly more complex and is dependent on the specific observation and the number of candidates found. In Figures 7.10 and 7.9, it can be seen that there is more variability in the run time of the components of the CO stage when scaling the N_t than when scaling $Z_{\max}$. In addition, it can be noted that for very short observations, there is some scaling of the CO components with N_t, however, beyond some point this scaling reduces significantly. This can be attributed to the candidates found; for very short observations, scaling N_t increases the sensitivity of the search, resulting in the detection of more candidates. However, beyond some point, the sensitivity is high enough to find the majority of the candidates, particularly the RFI candidates, at which point increasing the observation length does not result in significantly more candidates.

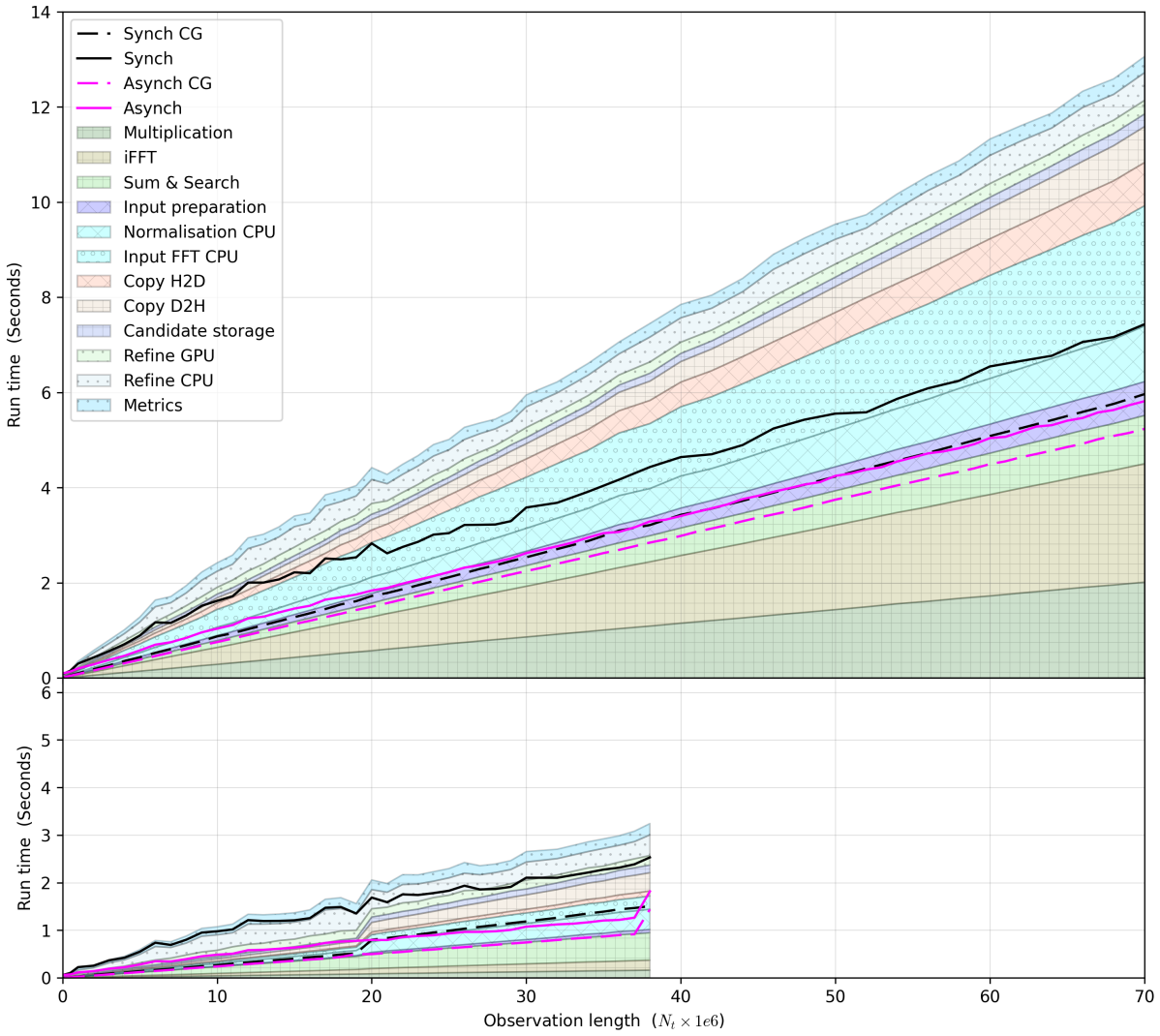


Figure 7.9: The scaling of run time with observation length, of the components of the archetypal search run on a GTX 1070. The component run times are shown by the shaded areas; with the synchronous and asynchronous run times indicated as the black and fuchsia lines. The run time of the CG stage is shown by a dashed line and the full search as the solid line, making the difference between the two the run time of the CG stage. The run time of all components scale roughly linearly with N_t; the run time of the CO components have a higher amount of variability than those of the CG stage. The run time of the asynchronous CG stage is very similar to the cumulative run time of the three critical GPU components of the stage: multiplication, iFFT, and SAS.

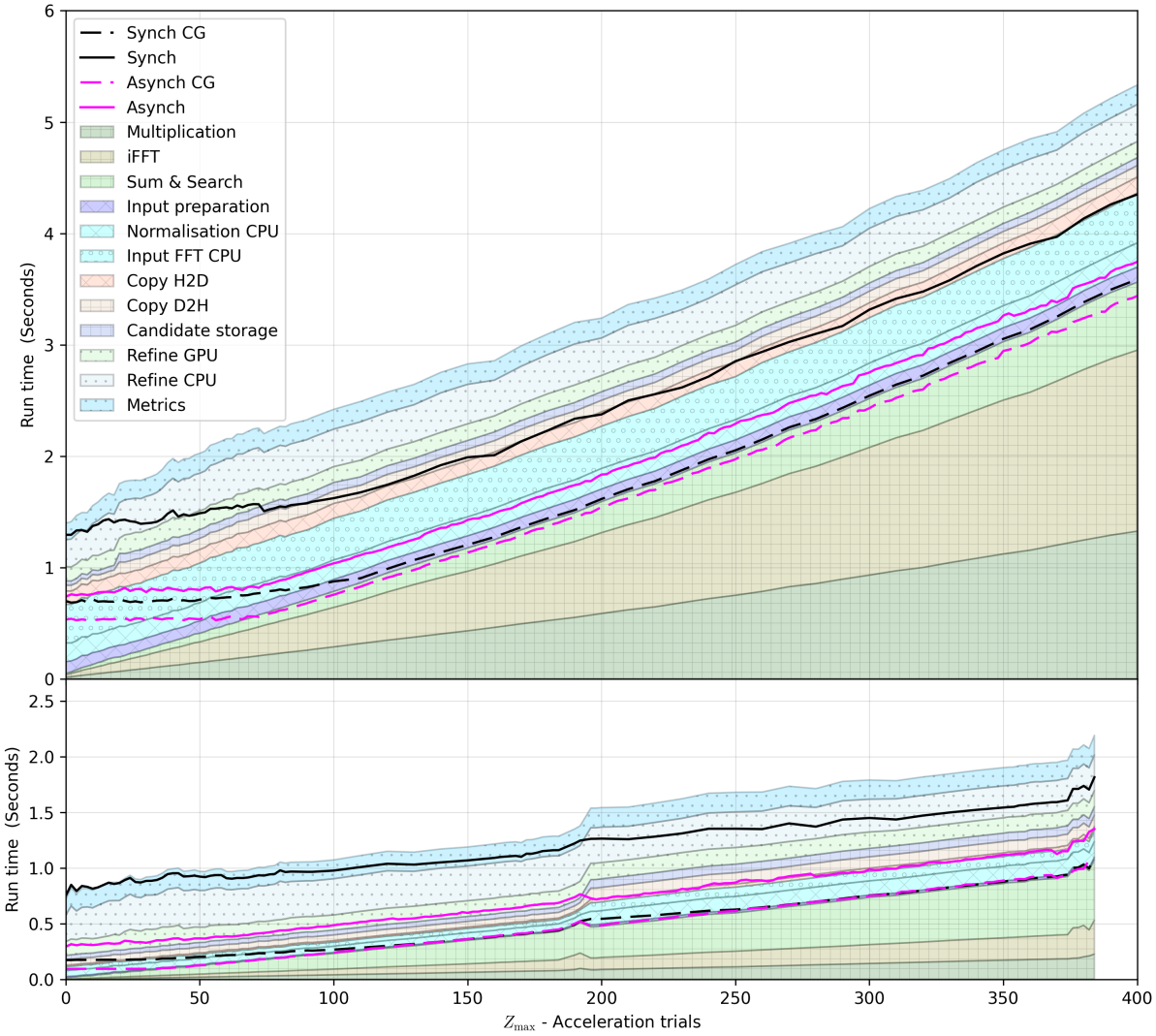


Figure 7.10: The scaling of run time with $Z_{\max}$, of the components of the archetypal search run on a GTX 1070. The component run times are shown as the shaded areas; with the synchronous and asynchronous run times indicated as the black and fuchsia lines. The run time of the CG stage is shown by a dashed line and that of the full search by the solid line, making the difference between the two the run time of the CG stage. In the CO component only the multiplication, iFFT, and SAS scale with $Z_{\max}$, and these three components dominate the run time for large $Z_{\max}$. The scaling of these components is not quite linear, the slight upturn is a result of the increase in contamination as $Z_{\max}$ is increased. The run time of the CO components have a higher amount of variability than those of the CG stage. The run time of the asynchronous CG stage is very similar to the cumulative run time of the three critical GPU components of the stage: multiplication, iFFT, and SAS.

7.3.2.2 $Z_{\max}$

Scaling $Z_{\max}$ changes the height of the sections of r-$\dot{r}$ planes created, thus a change in $Z_{\max}$ will affect the components relevant to plane creation: multiplication, iFFT, and SAS. Figure 7.10 shows the main components of the archetypal search, run with $\omega = 4096$ on the GTX 1070. To maintain consistency across the range of $Z_{\max}$, all searches used CPU input normalisation and FFTing of the input sections.

Figure 7.10 clearly shows how the run time of the multiplication, iFFT and SAS components scale with $Z_{\max}$ while the other components of the CG stage have fairly constant run times. This scaling is approximately linear and the three components scale at a similar rate, maintaining their relative proportions in both the standard and in-memory searches. The one exception is a slight upturn at the end of the in-memory search, which will be discussed in Section 7.4.3.

The three components of the CO stage – GPU location refinement, CPU location refinement, and metric calculation – can be seen as the top three components in Figure 7.10. The CPU refinement has some variability, especially for low $Z_{\max}$, but overall the three remain fairly constant over the entire $Z_{\max}$ range. As before, it is observed that the vast majority of the RFI-related candidates have an acceleration close to zero; thus, increasing the $Z_{\max}$ of the search does not produce many more candidates, which further results in the relatively constant run time of the CO components.

It is clear that the main difference between the standard and in-memory searches is the run times of the components of the CG stage. The largest differences are in the run time of the multiplication and iFFT components. As discussed previously, these differences are similar to the ratio of the number of r-$\dot{r}$ points (7.3), and remain fairly constant through the full range of $Z_{\max}$. The normalisation and input FFT components, as well as the various memory copies, differ between the two variants, while the SAS and all the components of the CO stage have very similar run times in both variants of the search.

The step at the midpoint of the in-memory components marks the transition between using a full and a split in-memory plane. This transition has little effect on the multiplication, iFFT, and SAS components, whereas it doubles the run time of components that deal with the input and output of the CG stage. When splitting the in-memory plane, the same sections of input data are used to generate both the top and bottom halves of the plane, essentially duplicating the

relevant calculations and memory transfers. Similarly, the outputs of the top and bottom halves are processed twice by the candidate storage component. This leads to the observed doubling in the run time of the relevant components. The affected components are generally not performance-limiting, thus this doubling has little impact on the asynchronous run time.

7.3.2.3 Harmonics summed

Figure 7.11 shows how the run time of the main components of the CG stage scale with h; which is roughly linear. In Section 7.1.5.1. we predicted linear scaling from the relative number of r-$\dot{r}$ points in each stage of harmonic summing; these ratios are shown by the grey lines in Figure 7.11. The H2D and the CPU FFT of the input data are slightly below the predicted values, while both forms of normalisation, as well as the GPU FFT of the input data are well below the predicted values. The scaling of the multiplication and iFFT components is very close to the prediction while that of the SAS component is slightly higher than the predicted values. These latter three dominate the run time of the CG stage, explaining why the scaling of the run time of the asynchronous CG stage of a standard search is so close to the predicted values.

In Section 7.2.1.1, we saw that splitting the in-memory plane decreases the run time of the in-memory search, and that this decrease is more severe when h is small. This decrease in speed is due to the H2D memory transfer taking longer than the SAS component. Splitting the in-memory plane means that the sum-and-search sub-stage will be run in two rounds, each will have CUDA kernels that run for half the time of their equivalents that search the full plane. Having two rounds of half-sized planes does not significantly affect the total run time of the SAS component (Figure 7.10). However, each split plane returns the same number of results, thus doubling the H2D memory transfer (Figure 7.10). When h is large, the run time of the SAS component is more than double the H2D memory when the plane is not split (Figure 7.8). Thus, for large values of h, splitting the plane does not significantly affect the asynchronous run time, as was observed in Section 7.2.2.2. However, it is evident from Figure 7.11 that the SAS and the H2D memory transfer scale differently with h. When h is small, splitting the in-memory plane can result in the H2D memory transfer being longer than the SAS component. The sum-and-search sub-stage only comprises three components: SAS, H2D memory transfer, and candidate storage. Thus, in some cases, the H2D memory transfer is the limiting factor and completely dictates the run time

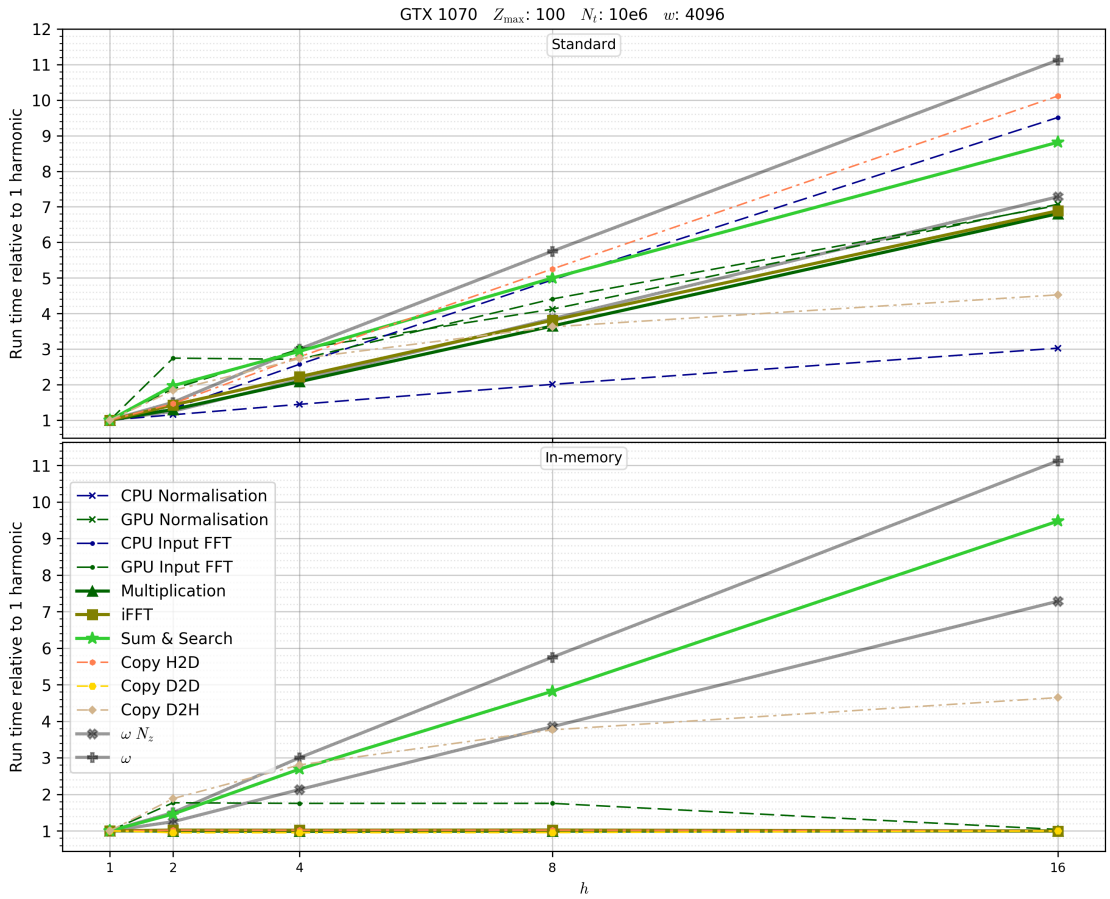


Figure 7.11: The scaling of run time with the number of harmonics summed, of the components of the archetypal search run on a GTX 1070. The ratio of the number of points in the relevant families of planes are shown by the grey lines; in Section 7.1.5.1 these were used to predict the scaling of the components. The scaling of the input components is slightly below the predicted values (ω), while the scaling of the multiplication, iFFT, and SAS components are much closer to the relevant prediction (ωN_Z). The run time of the asynchronous CG stage is very similar to the cumulative run time of the three critical GPU components of the stage: multiplication, iFFT, and SAS.

of the search sub-stage. This is particularly noticeable when h or $Z_{\max}$ are small, as these both affect the SAS kernel but not the H2D memory transfer. Most searches try to maximise $Z_{\max}$ and h, thus this problem affects a noncritical region of the search parameter space.

7.3.3 Asynchronous execution

The run times of the search run in both the asynchronous and synchronous modes are overlaid as lines in Figures 7.10 and 7.9. The dashed lines are the run times of the CG stage, and the solid lines represent the run times of the full search, making the difference between the two equal the run times of the CO stage. The synchronous search is shown in black, and the asynchronous search is shown in fuchsia. These can be used to highlight a number of the features seen in the standard asynchronous search.

The run time of the full synchronous search – the solid black line – is less than the cumulative sum of all the components; this is due to the synchronous search not truly being fully synchronous. As discussed in Section 6.2.4.4, the GPU computation, CPU computation, and memory copies between the host and the device are all run synchronously with respect to themselves, but may overlap with each other, allowing three-way concurrency. This first level of concurrency can be seen to significantly reduce the run time, especially in the standard search, and as $Z_{\max}$ increases.

Candidate generation

First, we examine the synchronous run times and then extend these results to the asynchronous case. For small $Z_{\max}$ ($Z_{\max} \leq 50$), the run time of the synchronous CG stage – the black line – is roughly constant. From $Z_{\max} \approx 50$, the run time begins to increase at the same rate as the components. The constant run time is roughly equivalent to the sum of the CPU components of the CG stage, which do not scale with $Z_{\max}$. Thus, the $Z_{\max}$ value of $\sim$50 marks the point at which the run time of the GPU computation, which scales with $Z_{\max}$, exceeds the constant run time of the CPU computation, this is discussed further in Section 7.4.2.

These synchronous results can be extended to the run times of the asynchronous search, shown by the fuchsia lines, to explain some of the results observed in Section 7.2. In the CG stage, there is a similar constant run time for small $Z_{\max}$. This constant section is, however, longer and at a lower value than its synchronous counterpart and is still attributed to the invariant CPU computation.

The lower value is a result of the CPU computation being run asynchronously across a number of CPU threads. In a standard asynchronous search, three CPU threads – each running a separate CG plan– are used in the CG stage; this allows the concurrent CPU normalisation and FFTing of three sets of input data. This concurrency reduces the run time of the full search, however, the reduction is not quite one-third as one may expect; this is due to administrative overhead and some "wasted" time caused by the synchronisation required between iterations in the asynchronous CG pipeline.

For larger $Z_{\max}$, the run time of the asynchronous CG stage – the dashed fuchsia line – is very similar to the sum of the three main GPU components: multiplication, iFFT and SAS. This shows that in the asynchronous CG pipeline, for large $Z_{\max}$, the limiting factor is the GPU computation. Thus, beyond some $Z_{\max}$ threshold, the run time of the CG stage is completely determined by the run time of the three critical GPU components. These three GPU components were the main focus of our optimisation efforts in the CG stage and are the primary subject of the next section. It is noted that the component run times shown here are the sum of hundreds to thousands of individual CUDA kernels run synchronously. The fact that the run time of the asynchronous CG stage is so similar to the sum of the synchronously run GPU components, shows that there is very little overlap between CUDA kernels in asynchronous execution.

Candidate optimisation

The synchronous run time of the CO stage – the difference between the solid and dashed black lines – is relatively large when compared to the cumulative run time of the components. This longer run time is a result of the components of the CG stage being run with very little overlap between CPU and GPU computation.

The asynchronous execution of the CO stage is slightly more complex. In Figures 7.10 and 7.9, it can be seen that the run time of the asynchronous CO stage is significantly less than that of the synchronous equivalent. In the synchronous case, the run time is dominated by the CPU location refinement component. However, in asynchronous execution, each CPU location refinement is performed by a separate CPU thread. Thus, if there are sufficient CPU cores, these computations can occur concurrently with each other and the GPU computation. Consequently, this CPU component does not significantly increase the run time of the asynchronous CO stage.

Unlike the CG stage, the run time of the asynchronous CO stage is not completely dominated by the GPU computation; the run time of the CO stage is approximately twice the duration of the GPU location refinement component, shown by the spotted green shaded region. In the case of Figures 7.10 and 7.9, this time accounts for approximately one-third of the run time of the asynchronous CO stage. The remaining time can be attributed to both the host-side administration required to perform the iterative GPU location refinement and the time required to spawn and manage the CPU threads used in the CO stage. We reiterate the fact that the CO stage run times shown here may be slightly higher than the average speeds that may be expected in a large-scale survey, due to the amount of RFI present in the observation used. However, this highlights the fact that for an in-memory search, the CO can account for a significant proportion of the total run time of the search, even for relatively large $Z_{\max}$.

Asynchronous pipeline

The results shown here highlight the value of exploiting parallelism at every possible level. In this work, our synchronous mode is in fact asynchronous, as it exhibits three-way concurrency, which is the maximum point to which a large degree of software is optimised. Our fully asynchronous pipeline allows us to decrease the run time over the standard three-way concurrency. This is achieved by using multiple CPU threads to perform the CPU computation, while at the same time asynchronously launching many CUDA kernels and memory copies. These allow greater utilisation of the CPU and the GPU. Take, as an example, the in-memory variant of the archetypal search: the synchronous run time is more than double that of the equivalent asynchronous search, validating the effort put into optimising our asynchronous pipeline.

7.4 Configuration parameters

The full search, as well as the individual components, have a large number of configuration parameters. Up to this point, the run times discussed have been largely obtained with the configurations of the search being decided automatically; the hope is that searches run in their default configuration will have close-to-optimal performance. However, achieving optimal performance is not a trivial task. Many parameters affect multiple components and there are many interactions between various parameters. Some configurations of common parameters will benefit certain components while impairing others. These relationships may differ between generations of GPUs

as well as between individual GPUs of the same generation. Examining all the combinations of configuration parameters for even a limited range of search parameters is unfeasible. In this section, we present a hierarchical examination of the most important configuration parameters and their effects on the dominant components. The component-specific effects were obtained by running the search synchronously; however, it is important to note that a configuration that decreases the run time of a component, may not decrease the asynchronous run time of an entire stage. Thus, we will often show the effects of configuration parameters on asynchronous run times.

7.4.1 Plane width

The plane width is the keystone configuration parameter of the CG stage, with many of the other parameters dependent on it. The input DFT is iterated over in segments, and S_w is determined from the ω parameter (Section 5.2). There are two factors that are influenced by the plane width. The first – the dominant factor – is how the run time of the individual components scale with plane width. The second, and lesser, factor is how the amount of contamination changes with plane width. These two factors are in turn influenced by $Z_{\max}$, h, and the hardware used in the search, thus the optimal plane width is somewhat dependent on these as well.

Components

The iFFT is the dominant factor in determining the optimal plane width, but the other two critical components have a lesser, though significant, impact. Figure 7.12 shows the run times of a number of the key components of the CG stage across several plane widths. The run time of the iFFT has a saddle shape and rises significantly for larger plane widths. In the standard search, this component has the highest run time; in the in-memory search, the SAS has the highest run time; however, the SAS run time is unaffected by ω. Thus, the primary component determining the optimal plane width is the iFFT. The steep rise in the run time of the iFFT for larger plane widths is due, in part, to the computational complexity of the iFFT, which results in a steep increase in the number of operations required to calculate the iFFTs as the plane width is increased. However, the run time does not follow the expected $N \log_2(N)$ scaling of the iFFT and differs between the generations of GPUs. Thus, a more likely explanation for the rise is as follows: as the number of elements increases, the 48 KB[3] of shared memory becomes insufficient to hold all the values to be Fourier

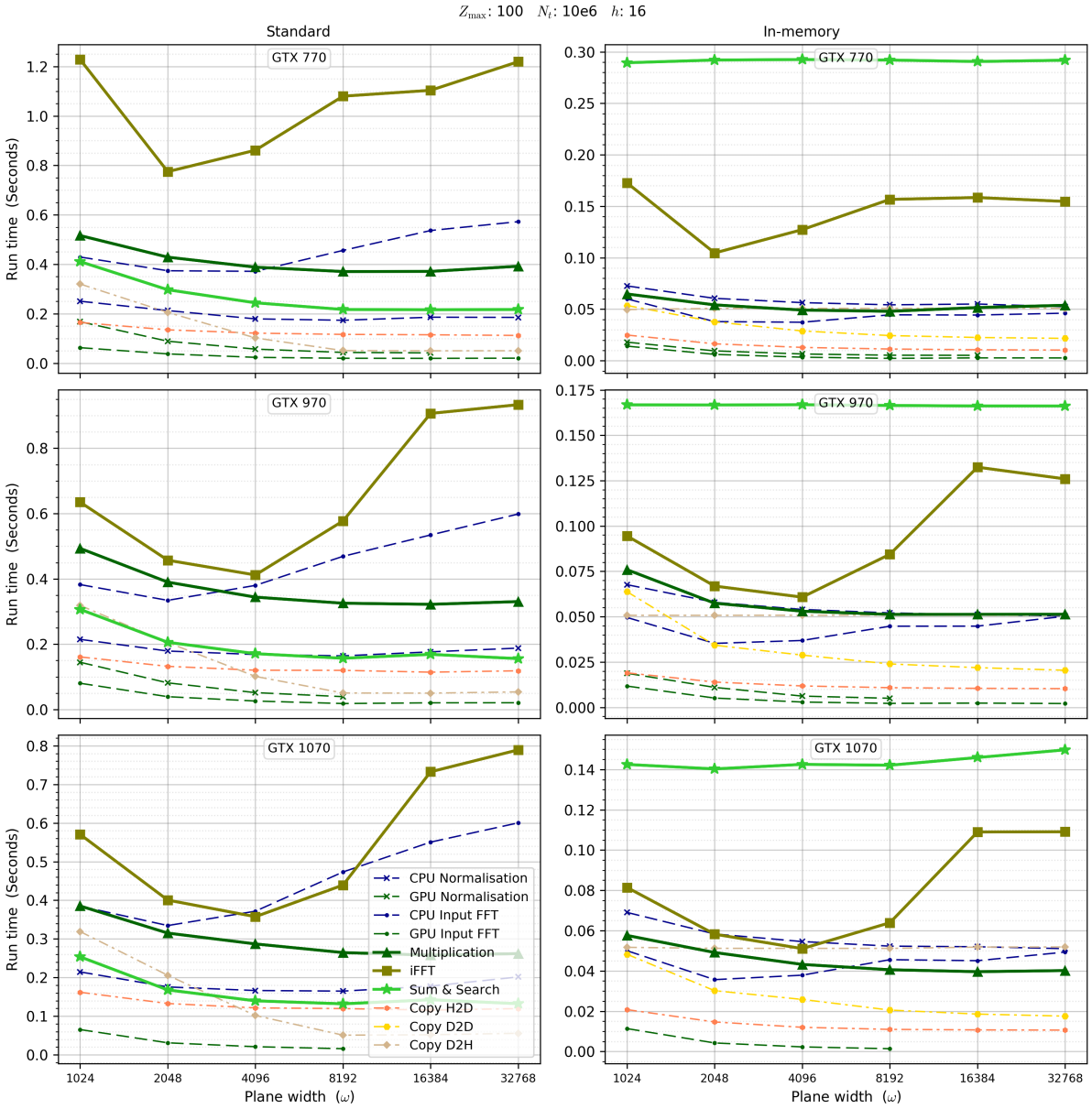


Figure 7.12: The run time of the CG components of the archetypal search over a range of plane widths. The run time of the iFFT component has a marked saddle shape and the severe increase for large ω puts an upper bound on the optimal plane width. The run time of the other two critical components decreases with ω. The optimal plane width is found where the cumulative run time of these three components is at its minimum. This optimal plane width differs across GPUs and other configuration parameters but is usually 4096 or 8192.

transformed. This conjecture is further strengthened by the fact that the older GTX 770 has a smaller amount of shared memory per SM. An additional factor may be the increased number of registers in the newer generations, which may significantly aid in the faster computation of FFTs in these generations. Irrespective of the cause, this significant rise in the run time of the dominant component places the upper limit on the optimal plane width.

Counteracting this preference for smaller plane widths, the run time of the other two critical components decrease gradually as plane width increases. This can be attributed, in part, to increased efficiency, as wider plane widths will result in fewer, wider planes being generated. Wider planes result in more CUDA threads, and thus more thread blocks, used in the multiplication and SAS kernels, which can increase device occupancy. The optimal plane width is found at the minimum cumulative run time of the critical components. It is noted that in the case of the asynchronous archetypal search, the optimal plane width is 4096 for all the GPU generations tested. In the case of the GTX 970 and GTX 1070, the optimal asynchronous plane width corresponds to the minimum run time of the iFFT component. However, this is not the case with the GTX 770, showing that the other two components do have some impact on the optimal plane width. It is important to note that, while the other components have similar trends between the three generations, the iFFT component differs: for this component the three GPUs have their minimum run times at differing plane widths. This highlights the premise that the optimal configuration parameters are dependent on the particular hardware used.

Contamination

The factor which places the absolute lower limit on the plane width is the amount of contamination (Section 3.2.1) present in the primary plane, which is directly linked to the $Z_{\max}$ of the search (Section 6.2.1). Increasing $Z_{\max}$ not only increases the time required to process a single batch, it increases the contamination and thus the number of segments into which the input DFT is divided. This results in a nonlinear scaling of run time with $Z_{\max}$ for a given plane width. When the width of the contaminated region is significantly less than the plane width, the scaling of run time with $Z_{\max}$ is close to quadratic but very gradual; however, it increases asymptotically towards infinity as the proportion of contamination rises. The top panel in Figure 7.13 shows the

[3] Storing 4092 single-precision complex values requires 36.8 KB of memory while storing 8184 complex values requires 81.9 KB.

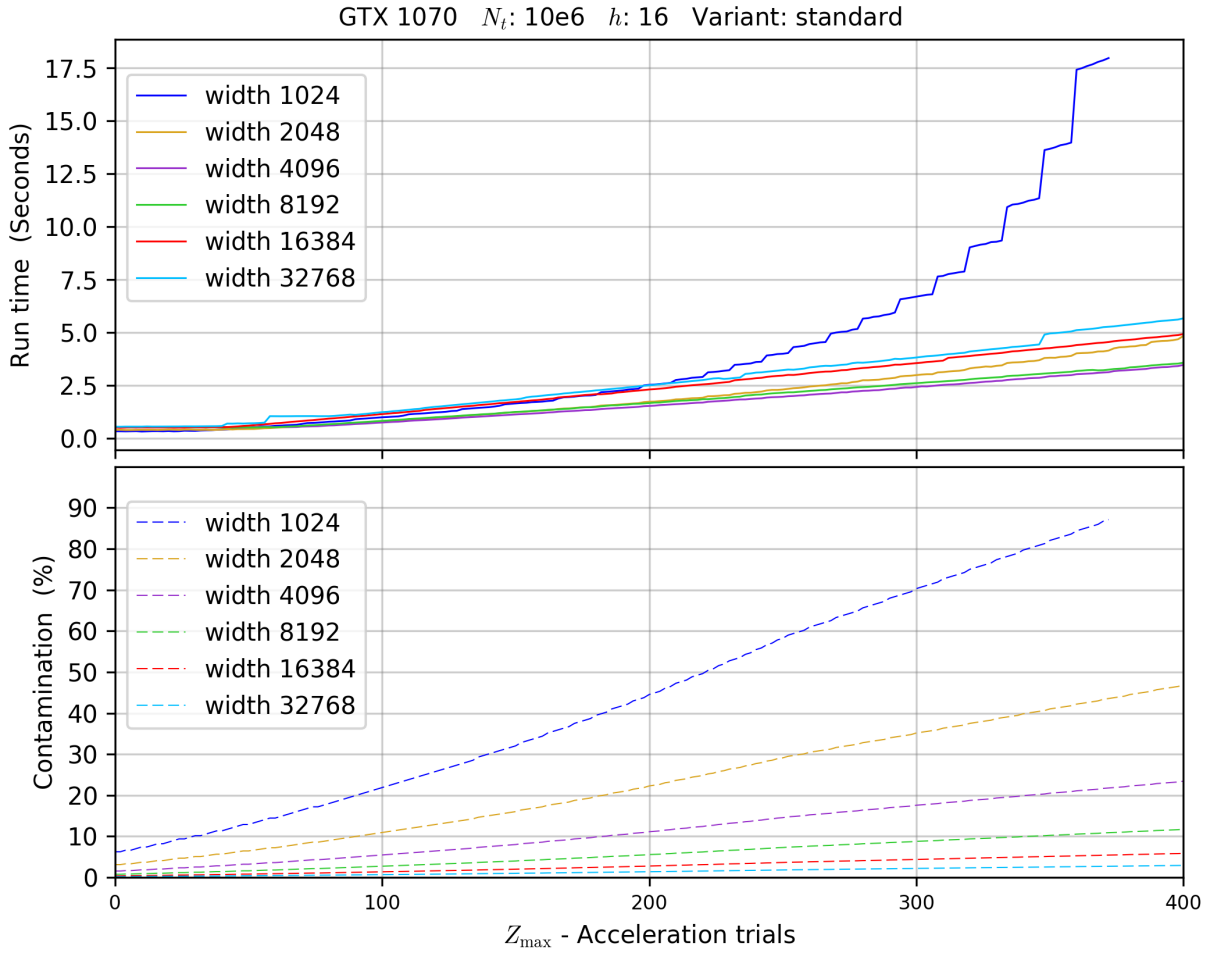


Figure 7.13: The run time of the CG stage of the archetypal search across a range of plane widths. Contamination causes nonlinear scaling of the run time with $Z_{\max}$. This is most noticeable when the plane width is small, in which case the contamination is a significant proportion of the primary plane. This contamination puts a hard limit on the smallest plane width, in the case of $\omega = 1024$, the run time will increase asymptotically to $Z_{\max} = 426$, from which point the primary plane will be completely contaminated. The optimal plane width is always well below the point where this asymptotic growth begins, we find the optimal plane width will usually have less than 20% contamination.

run time of the asynchronous CG stage for a number of plane widths, across a range of $Z_{\max}$. The asymptotic increase in run time is clearly visible in the narrowest plane width (1024), which has its asymptotic bound at $Z_{\max} = 426$. The beginning of this asymptotic growth is evident in the next narrowest plane width (2048), and is hardly noticeable in the wider plane widths. The steps in the run time are a result of the segment size only being changed at discrete points, so that the start of each segment is aligned on a multiple of h, a requirement of the standard variant of the SAS kernel (Section 6.2.3.6).

The subtle effects of the change in contamination are visible in a number of the previous figures in this chapter. In Figure 7.10, a fixed plane width of 4092 was used. With this plane width, the contamination in the primary plane starts at 1.56% at a $Z_{\max} = 0$, and increases to 23.44% at $Z_{\max} = 400$. This slow increase can be seen to cause a very slight uptrend in the run time of the standard variants of the multiplication and iFFT components, this is the slow quadratic growth mentioned above. This nonlinear scaling causes the speed of running the CG stage to gradually decrease as $Z_{\max}$ grows large, this is particularly noticeable in Figure 7.3 especially when summing few harmonics. In Figure 7.12, the only factor affecting the H2D memory transfer is the amount of contamination, thus the increase at smaller plane widths is caused solely by the increased proportion of contamination. The effects in all these cases are minor, however, as $Z_{\max}$ increases they will become more significant. Contamination limits the minimum plane width, but the point at which the optimal plane width changes is well below full contamination. We find that contamination of more than 20% of the primary plane is undesirable and generally marks the point at which to transition to a wider plane width.

From these results, we conclude that the main factor contributing to the optimal plane width is the combined run time of the three critical components of the CG stage, which are dominated by the iFFT, with the amount of contamination playing a secondary role. It is challenging to generalise optimal plane width as it is dependent on the hardware and iFFT implementation. However, for most searches, a plane width of either 4092 or 8192 is optimal, with 2048 occasionally being optimal when summing lower numbers of harmonics or when using older GPUs. We recommend keeping the amount of contamination in the primary plane below 20% and using a plane width of 4092 as an initial or default choice.

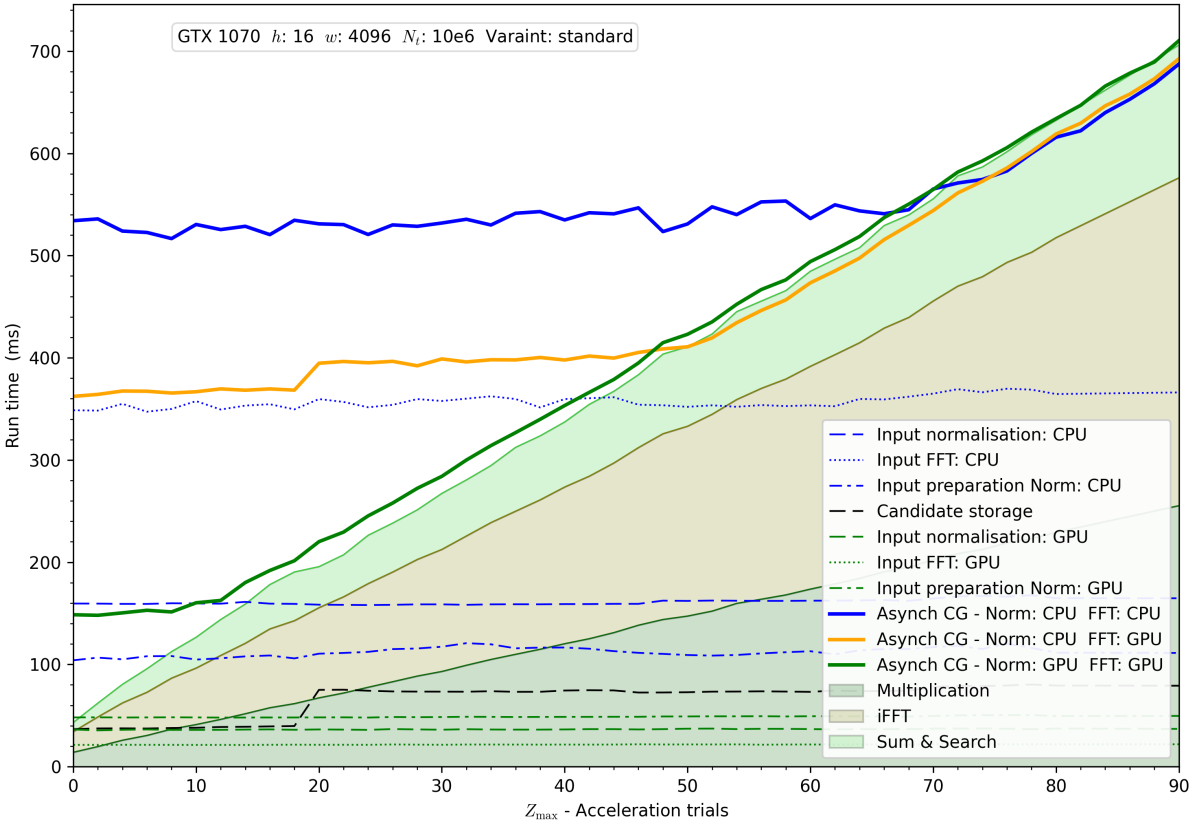


Figure 7.14: Run time of the CG stage and of the components, for small $Z_{\max}$. The run times of the asynchronous CG stage using the three possible ways of distributing the normalisation and FFT components between the two processing units are shown as the thick lines. For small $Z_{\max}$, using the GPU to perform these components results in the lowest asynchronous run times. However, for larger $Z_{\max}$, these components can be performed by the CPU concurrently to the GPU computation of the other components of the stage, resulting in a lower asynchronous run time.

7.4.2 Distribution of work between the CPU and the GPU

The normalisation and FFT components can be performed by either the CPU or the GPU and thus – to some extent – allow the distribution of work to be balanced between the two processing units. These components deal with the input used to generate the r-$\dot{r}$ planes, and thus their run time does not scale with $Z_{\max}$. In Figure 7.14, the thick solid lines show the run times of the asynchronous CG stage, using the three possible ways of distributing the two components between the CPU and the GPU. These all start with some roughly constant, but significantly different, run time. However, once these constant run times intercept the cumulative run time of the three critical components – shown as the shaded, stacked area – they begin to scale with

$Z_{\max}$ at the same rate as the critical components. This shows that for small $Z_{\max}$, the run time of the CG stage is determined by the components that do not scale with $Z_{\max}$. However, from some point, the run time of this stage is set by the three critical components which do scale with $Z_{\max}$. For small values of $Z_{\max}$, using the faster GPU to perform both input components – the solid green line – results in the fastest run time. However, for larger $Z_{\max}$ values, performing these components using the slower CPU results in the fastest asynchronous run time. From this point, the computation can be moved to the slower, but underutilised CPU, taking a small but noticeable load off the GPU, which decreases the run time. Thus, for small values of $Z_{\max}$, using the GPU to perform all the computational components gives the best run time. However, for larger $Z_{\max}$, some of the smaller computational components can be shifted to the CPU, decreasing the overall run time. To accommodate this, there are two parameters in the configuration text file which control the $Z_{\max}$ values, beyond which the search will switch between GPU and CPU normalisation and input FFTs.

The run times of some components are shown as the various dashed and dotted lines in the figure. We note that in the regions where the asynchronous run times are flat, the combined run time of the components run on the CPU are very similar to the constant run time of the asynchronous CG stage. However, when running asynchronously, three CPU threads were used to run the CPU components, thus one may expect the asynchronous run time to be a third of the combined run time of the components. This shows that, in this range, the asynchronous pipeline is not operating very efficiently; and that there is potential to optimise the asynchronous behaviour to better exploit the CPU. The focus of this work was on the optimisation of searches of higher $Z_{\max}$, and the GPU elements of the pipeline. Thus, in this range, the relevant components can simply be performed on the GPU to maintain the close-to-linear scaling of run time with $Z_{\max}$, right down to very low values of $Z_{\max}$.

7.4.3 Number of CG plans and segments per CG plan

The computational complexity of the iFFTs used in the CG stage keep the optimal plane width relatively narrow. Reducing the plane width reduces the size of the sections of data worked on by the components. Having small units of work may negatively impact efficiency and reduce the ability to hide instruction and memory latency in CUDA kernels. Our primary method of

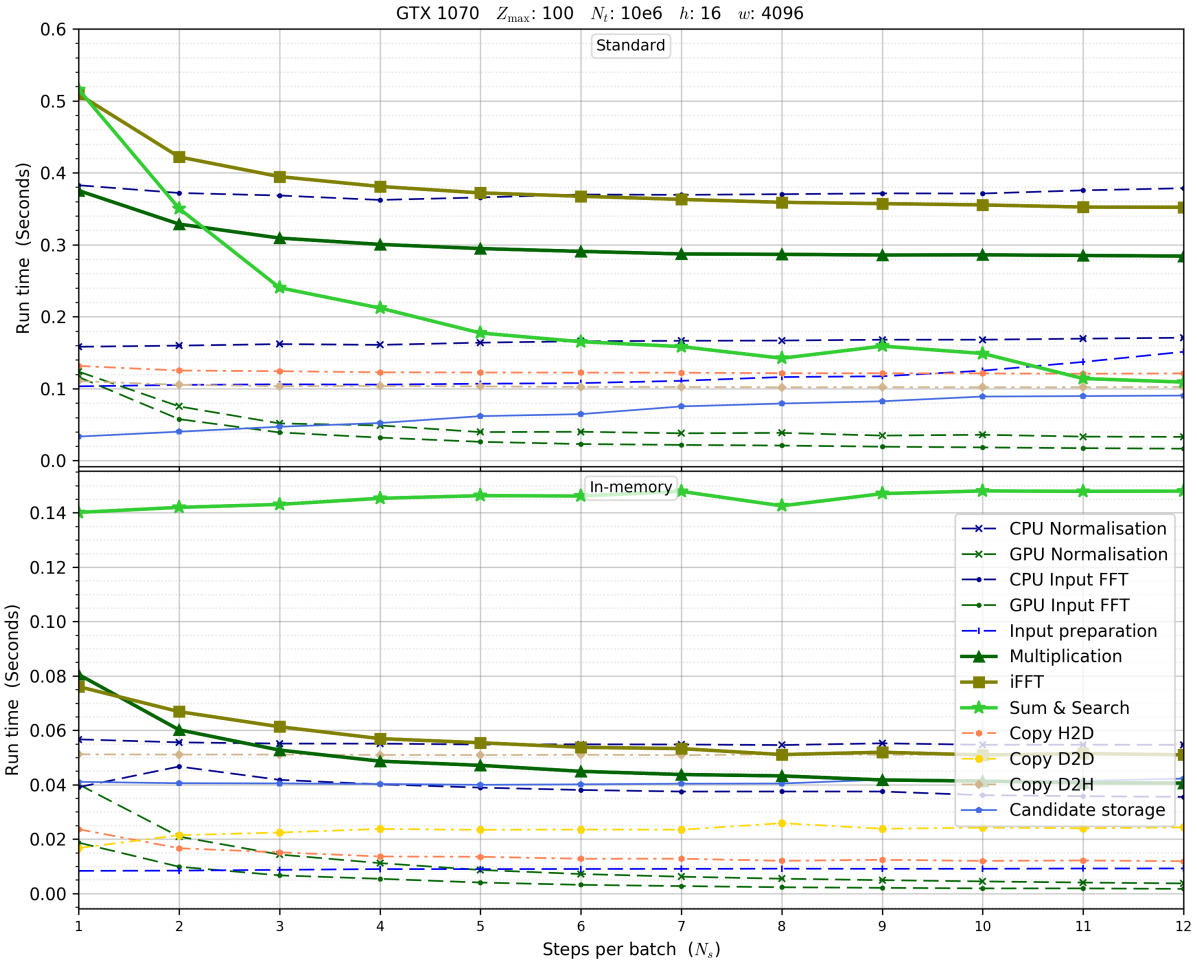


Figure 7.15: The run time of the components of the archetypal search over a range of N_s. Increasing N_s decreases the run time of the three critical components. These results were obtained using a single CG plan run asynchronously.

combating this is by aggregating N_s adjacent segments into a single batch. This allows many kernels to operate on larger sections of data and amortises some computation and memory transactions. Using a batch means that all the components of a CG plan have to work on data with the same number of segments. The number of segments affects each component differently, and this effect is influenced by the GPU architecture. Figure 7.15 shows the run time of a number of components over a range of N_s values. Increasing N_s has a trend of decreasing the run time of the three critical components: multiplication, iFFT, and SAS. The most significant decrease is seen in the standard variant of the SAS component, where the run time can be decreased by up to two-thirds. Increasing N_s increases the run time of a number of CPU components, due to their greater complexity. However, as was demonstrated in the previous section, for large $Z_{\max}$ values, CPU computation is not the factor limiting asynchronous run time. The factor limiting run time

is the cumulative run time of the three critical components, which decreases as N_s is increased. We note that, as was shown in Sections 6.2.3.3 and 6.2.3.6, increasing N_s too much can increase register use and cause register spilling. It is observed that, in general, a value of N_s between six and eight gives the best performance across a wide range of configuration and search parameters.

Each CG plan is associated with a separate CPU thread and the GPU computation of each is largely independent of the other plans. The use of multiple, concurrently running plans allows greater asynchronous CPU computation and may increase GPU occupancy. In `AccelGPU`, N_s is common to all the CG plans used in a given search. It was found that using multiple plans is generally advantageous, with $N_g = 3$ generally performing very well.

When device memory is limited, either N_g, N_s, or both will have to be reduced from their optimal values. Each plan requires some working memory on both the host and the device. The amount of device memory needed per CG plan scales linearly with $Z_{\max}$, ω, N_s, and – in a standard search – h. As a reference, in a standard search with $\omega = 4096$, $h = 16$, $Z_{\max} = 100$, and $N_s = 8$, each CG plan requires 248 MB of device memory; this drops to only 37 MB in an in-memory search. Figure 7.16 shows the search rate of the standard variant of the CG stage, across a range of N_g and N_s. The most naïve implementation – that of $N_s = 1$ and $N_g = 1$ – has the worst performance across the board and can take up to twice as long as the optimal combination, which is usually a balance of the two. This highlights the value both of using multiple CG plans and of aggregating multiple segments into a batch. The default combination of the two parameters is $N_g = 3$ and $N_s = 8$; this combination gives good performance across a wide range of both search and configuration parameters. As is evident in Figure 7.16, it is generally preferable to have $N_g > 1$ and $N_s > 2$; however, when device memory is limited, one or both of these has to be reduced. When these parameters must be reduced below the preferable ranges, it has been observed that if h is low, a greater number of segments is desirable, while if h is high, selecting more CG plans gives faster run times. As demonstrated in the previous section, if the $Z_{\max}$ is low, the factor limiting performance can be the CPU computation, thus – in these cases – run time could be reduced by increasing the number of CPU threads. Therefore, when $Z_{\max}$ is low, increasing N_g rather than N_s is preferable.

In Figure 7.16, the blank combinations mark the points where the device memory was insufficient to store the combination of N_g and N_s. Limited memory occurs most often when performing an

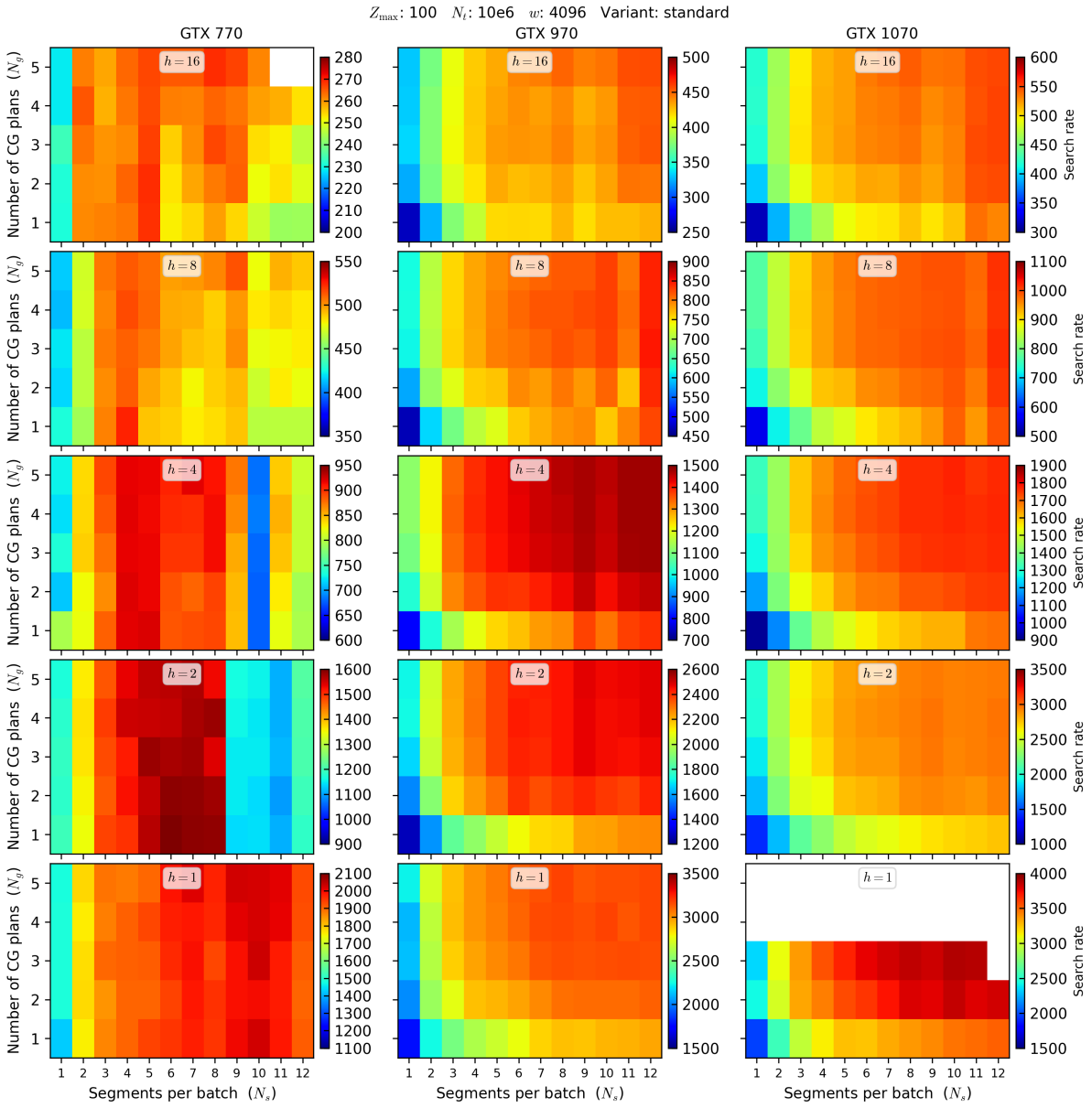


Figure 7.16: The search rate of the asynchronous CG stage across a range of N_g and N_s. The N_g and N_s parameters allow some control of the granularity of the CG stage. Increasing N_g allows greater utilisation of both the CPU and the GPU, while scaling N_s allows more efficient CUDA kernels and the amortisation of some memory transactions. However, increasing N_s raises register use and may cause detrimental register spilling as can be seen on the GTX 770 with $h = 2$. The amount of working memory needed by the CG stage scales with the product of the N_g and N_s parameters, the blank area in the top left diagram indicates regions where the required working memory exceeds the device memory.

in-memory search. As the in-memory r-$\dot{r}$ plane grows, there is less device memory available for the plan-specific work areas, thus N_g and N_s have to be reduced. This will usually be done automatically by the application. The effects of this automatic scaling of N_g and N_s can be seen in a number of the plots in this chapter. In Figures 7.3 and 7.2, the speed of the in-memory searches can be seen to drop off just before the in-memory plane is split and again before the end of the in-memory domain. This is a result of the need to use a suboptimal combination of N_g and N_s. A similar behaviour is evident in the run times of the asynchronous, in-memory CG stage in Figure 7.10. The variability in the run time marks the points where the N_g and N_s are changed. In Figure 7.10, the profiling of the components is done with a single plan, usually with $N_s = 8$. However, N_s is gradually reduced as the in-memory r-$\dot{r}$ plane approaches its maximum size. This causes the marked upturn in the run times of the three critical components towards the end of the in-memory range. Finally, in Figure 7.13, there are a number of steps in the Z-speed, especially noticeable with wider plane widths at high $Z_{\max}$ values. These steps are caused by the automatic changes in the distribution of N_g and N_s.

7.4.4 Slices

The multiplication and SAS CUDA kernels allocate a thread to each column of a stack. Thus, if the plane width is small, the number of threads created for a single kernel invocation will be low; this is especially problematic in the case of the multiplication kernel, where the smallest stack may be an eighth of the width of the largest stack. CUDA kernels with few threads can be detrimental to performance as this can decrease the ability to hide the instruction and memory latency. This problem is addressed by allowing the stacks and planes to be "sliced" into independent horizontal slices, with threads now being allocated to columns of slices, as described in Section 6.2.3.3. This slicing allows the number of CUDA threads to be increased, at the cost of a small number of duplicated calculations or memory accesses.

It is generally advisable to have 512 or more threads per SM. Thus, if we assume the number of SMs to be in the tens, one may expect some advantage in slicing stacks narrower than 8192. If run synchronously, we do see some significant improvement when slicing some kernels, especially when summing a large number of harmonics. In the standard variant of the SAS kernel with $\omega = 2048$, we see a reduction in the run time of up to 64% when splitting the kernel into 5 slices.

Furthermore, the improvements in the multiplication kernels range from 24% to 40% from the widest to the narrowest stack. The improvements in the in-memory search are minor; the width of the SAS segments are independent of the plane width and are thus specifically chosen to be much wider. Therefore, slicing has little effect in these cases. There is some small advantage to slicing the multiplication kernels in an in-memory search, but this is generally less than 5% for plane widths of 2048 or greater.

The results above were obtained by running the components synchronously, meaning that each CUDA kernel was run in complete isolation. However, in normal asynchronous execution, there should be many independent CUDA kernels running, or queued to run. If a CUDA kernel has few threads, and thus warps, it can be run concurrently on a single SM with the warps from other CUDA kernels. Thus, in asynchronous execution, other kernels may hide some of the memory latency. As a result, the performance improvements of slicing are significantly lower when running asynchronously. In the asynchronous, standard variant of the CG stage, the improvements are between 1.5% and 8%. Interestingly, the best improvements are obtained when summing fewer harmonics; there is an approximate 8% improvement when summing either one or two harmonics, which is encouraging since in this case, the standard search is preferred over the in-memory search. The improvements for the asynchronous in-memory search are generally less than 0.5%.

The marked differences between the effects in synchronous and asynchronous execution, serve as a good example of how optimisation in isolation may not translate to a significant improvement in the asynchronous run times. Nonetheless, it has been observed that slicing – when used within reasonable bounds – never detrimentally affects asynchronous run times. Under circumstances where only one CG plan with a potentially limited number of segments must be used, slicing may compensate for a decrease in occupancy caused by the reduction in the number of asynchronous CUDA kernels. Thus, we concluded that slicing is recommended when working with narrow stacks. By default, all kernels – where slicing is implemented – will be sliced so that there are at least 8192 threads.

A subtle side-effect of the automatic slicing can be seen in Figure 7.8. The plane width used for those results was 4096, meaning that the SAS kernel used in the standard search was divided into two slices and, as described in Section 6.2.3.6, each slice returned a “complete” set of initial candidates. By contrast, in the in-memory search, only a single slice is used in the SAS kernel,

thus only one set of initial candidates is returned for each SAS kernel. The run times of the memory copies from the device are therefore double in the case of the standard search; similarly, this approximately doubles the run time of the CPU candidate processing component, which deals with these results. For searches with large $Z_{\max}$ values, this doubling does not make a noticeable difference in the normal asynchronous search, as neither component limits the asynchronous run time.

7.4.5 Powers

The powers are calculated in a post-transform callback to the iFFT, allowing them to be stored as half-precision floating numbers. Using this callback adds only a single 32-bit arithmetic operation and a conversion to half-precision, for each point in an r-$\dot{r}$ plane. This requires minimal computation and reduces the amount of memory written by the iFFT by half, or by three quarters in the case of storing half-precision powers. The additional computation has far less impact than the reduction in memory transactions, thus it would be expected that the post-transform callback would reduce the overall run time of the iFFT. Observation shows that this is not always the case; in the newer GTX 1070, the run time is indeed decreased by 25%; however, in the older GTX 770, this increases the run time by 20%. As was noted in Section 6.2.3.3, this may be due to the overhead required to simply run a callback function. However, an additional impact of using the callback is that the amount of memory read by the SAS kernel is reduced by three quarters; a significant consideration in a memory-bound kernel. This can reduce the synchronous run time of the standard variant of the SAS kernel by up to 50%. In a standard search across all values of h, calculating the powers as a callback and storing them as half-precision powers reduces the combined synchronous run time of the iFFT and SAS components by approximately 30%. As these are two of the three critical components of the CG stage, this results in a 20% to 25% reduction in the run time of the asynchronous CG stage. For the in-memory searches, the improvement is even greater, with the asynchronous CG stage of in-memory searches showing a 40% to 50% decrease in run time when using a callback to store half-precision powers. This is due to the SAS kernel being the dominant component of the in-memory CG stage. Consequentially, using cuFFT callbacks and half-precision powers is the default behavior.

7.4.6 Candidate storage

The candidate storage component is run on the CPU, and iterates through the results returned from the GPU. If a summed power is above the detection threshold, its sigma value is calculated; if this is greater than all spatially similar candidates, the candidate is stored, potentially displacing subsidiary candidates. The run time of this component can be highly variable and is dependent on the number of, and the location of, candidates found.

This component exemplifies the notion that, when previously dominant components are accelerated, components that previously were minor, such as this one, may become the element limiting run time – as is the case here. Thus, to accelerate an entire process it is often necessary to optimise even the smallest and seemingly insignificant components.

The run time of a single instance of the candidate storage component can be significantly greater than the run time of all other components of that iteration. If run synchronously, this can block other computations and easily double or triple the run time of the CG stage. To mitigate the effects of the highly variable run time of this component, a separate CPU thread is spawned to run the candidate storage component of each iteration; this allows a few prolonged instances of this component to run in the background while the rest of the search continues to be processed.

It was found that storing the candidates in a singly linked list, as is the case in the existing CPU implementation, is inefficient. On inspection, it was established that even the mere process of repeatedly iterating through the list can take an unacceptably long time when many candidates are found, even when run in a separate thread. Thus, `AccelGPU` stores the candidates in an array. This storage method has far faster access times and thus alleviates the inefficiency of using a singly linked list.

In a similar manner to the CPU processing of the input components (Section 7.4.2), it was found that – in the case of small $Z_{\max}$– the run time of this component can become significant. The extent of this is such that simply copying the results from pinned host memory to thread-specific memory can become the performance-limiting factor. This additional load on the CPU can mean that the input components cannot be run by the CPU. To alleviate this, in-kernel candidate counting (Section 6.2.3.6) was implemented; this counts and stores the number of initial candidates found by the GPU SAS component along with the actual results. Thus, the number of candidates

found by the GPU component of a CG plan can be determined by the host simply reading a single value. If no candidates are found, the returned results can simply be ignored, negating the need for the host to copy memory or spawn a CPU thread. Similar to the input components, it was found that in-kernel candidate counting can make a significant difference when the $Z_{\max}$ is small. Candidate counting uses shared memory and atomic operations in the SAS kernel. It was found that the act of simply allocating a small amount of shared memory drastically reduced the run time of the SAS component on Kepler GPUs. Therefore, candidate counting is automatically disabled on older GPUs, as increasing the run time of the critical SAS component has a larger impact than decreasing the run time of the candidate storage component. The effect on the candidate storage component can be seen in Figure 7.8, where the run times are more than double in the case of the GTX 770.

7.4.7 Location refinement

The CO stage has two components: location refinement and metric calculations. The former is the dominant component and is performed in a hierarchical fashion by both the GPU and the CPU. It begins by generating a low-resolution, low-accuracy r-$\dot{r}$ plane on the GPU, followed by generating some r-$\dot{r}$ planes with higher accuracy and higher resolution on the GPU, and concludes by performing a high-accuracy, double-precision Nelder–Mead refinement run on the CPU. This localisation is highly configurable, with the main parameters being the number of levels of plane generation and the dimension, accuracy, precision and resolution of the planes at each level. The nature of the final high-accuracy, double-precision refinement will play some role too. In the default configuration, four levels of GPU planes and a final double-precision CPU refinement are used.

A full analysis of all the combinations of the parameters is beyond the scope of this work. Rather, we show the performance of the default configuration and two alternatives – one weighted toward more GPU computation, and the other toward more CPU computation. The first configuration shown in Figure 7.17 is the default, as described in Table 6.1. The second configuration examined is one in which the entire location refinement, including all the double-precision processing, is performed on the GPU. The rationale for this is that, in the default configuration, the location is refined to practically the highest possible single-precision resolution using the GPU. Thus, to

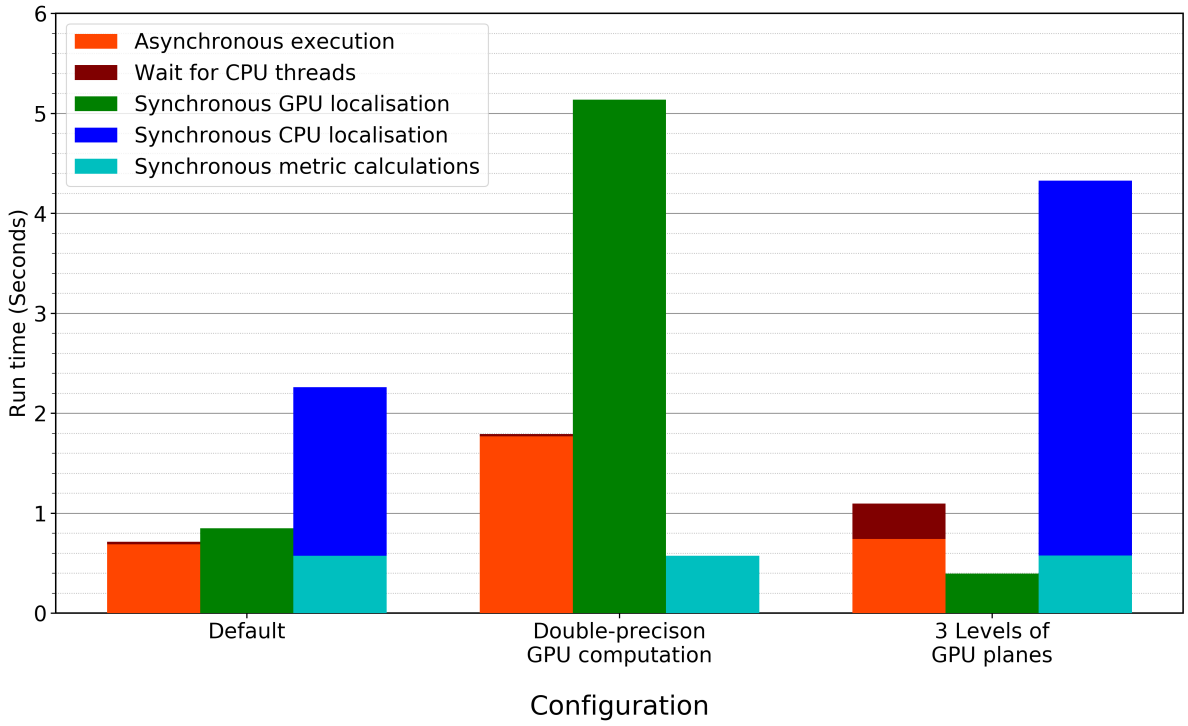


Figure 7.17: The run time of the CO stage of 222 real-world candidates over three differing combinations of parameters. These initial candidates are the output of the CG stage of a search of real data summing 16 harmonics. The first default combination gives the lowest asynchronous run time as it balances the computation between the CPU and the GPU. The second combination shifts some of the double-precision calculations to the GPU, this significantly increases the run time of the GPU components and thus the asynchronous run time. The last combination reduces the level to which the GPU is used to refine each maximum, this increases the amount of location refinement that is required to be performed by the CPU. Even when run asynchronously using many CPU threads, this increases the overall run time.

include more GPU computation requires performing some double-precision calculation on the GPU. However, the total amount of double-precision computation required to complete the entire location refinement is minimal and thus the entire double-precision refinement is performed on the GPU. This was done by including two additional low-dimension (16×16) double-precision levels, giving a similar accuracy to the default configuration (Section 6.3.4). In the third configuration, the number of levels of the GPU optimisation are reduced by 1, performing only the first 3 levels of the default configuration, followed by the standard double-precision Nelder–Mead refinement on the CPU.

As discussed in Section 6.3, the optimisation of the numerous candidates is run asynchronously by a number of CPU threads. The optimisation of a single initial candidate is done by running

an iterative GPU localisation and then spawning a CPU thread to perform the CPU localisation and final metric calculations. Figure 7.17 shows both the synchronous and asynchronous run times of the three configurations, used to optimise the same 222 initial candidates discussed in the validation of the method (Section 6.3.4). The first orange and maroon column in Figure 7.17 shows the run time of the standard asynchronous execution of the entire CO stage. The orange section shows the time taken to iterate through all the candidates, which comprises running the GPU components and spawning the threads to run the CPU computation. The maroon section is the portion of the run time of the CO stage spent waiting for the completion of all the spawned threads. The green column is the run time of all the GPU components run synchronously, while the last blue and cyan column is the synchronous run time of the CPU components run in the spawned threads.

In the default configuration, the total synchronous CPU computation is significantly greater than the total asynchronous execution and very little time is spent waiting for the CPU threads to complete once the asynchronous execution is complete. This shows that the CPU computation is distributed across the CPU cores, and run concurrently with the GPU computation. When only three levels of GPU optimisation are used, the CPU computation time is significantly increased and, importantly, in the asynchronous execution, a large amount of time is spent waiting for the spawned threads to complete. This shows that the run time of this variant is limited by the CPU location refinement. This is an indication that maxima have not been resolved to a sufficient level by the GPU location refinement, hence the additional level in the default configuration.

Doing the double-precision localisation on the GPU significantly increases the synchronous GPU computation. The dimension of the double-precision planes is very low – only 16×16 – thus, these kernels do not have sufficient threads to saturate the GPU. Therefore, when run synchronously, there are times when the GPU is underutilised, while in asynchronous execution multiple concurrent CUDA kernels are launched and can run concurrently, increasing occupancy and more-than-halving the run time. However, on the GeForce range of cards, this double-precision computation runs $\sim$32 times slower than the single-precision equivalents. Thus, the addition of even this very limited double-precision computation almost doubles the final asynchronous run time. This may not be the case if the GPU used has a greater proportion of FP64 cores per-SM, as many of the Tesla or Quadro cards have. Even in those cases, however, if the limited double-precision

computation can be run concurrently on the CPU, this may well be faster than doing it on the GPU.

In both the default configuration and the three-plane configuration, the asynchronous run times are similar, while the synchronous GPU computation differs. Thus, a reduction in the GPU computation does not translate to a significant reduction in the asynchronous run time, showing that this is not completely determined by the GPU computation. Rather, the administration of coordinating the GPU execution and launching the CPU threads plays an important role in the final asynchronous run time. This highlights one of the largest issues with the GPU acceleration of the CO stage. The optimisation of a single candidate involves a relatively small amount of computation – not so small that it cannot be significantly accelerated, but small enough that Amdahl's law comes into play, limiting the speed-ups attainable. In other words, the maximum possible speed-up though parallelisation is ultimately limited by the proportion of serial computation. It was noted that a large proportion of the administrative time is attributed to allocating and copying thread-specific host memory, as well as spawning the threads. In large multi-DM searches, many acceleration searches will be run; in such a scheme, there is the potential to run the CPU optimisation components of one search during the CG stage of another search, which may allow further reduction of the overall run time.

In `AccelGPU`, the planes created on the GPU contain orders of magnitude more r-$\dot{r}$ points than those used in the Nelder–Mead method, allowing an increase in the accuracy of the localisation. Thus, with the use of a GPU, it is possible to scale up the parallel computation to a point where it is commensurate with the serial computation. It is thus possible to improve the performance to a close-to-optimal level, and at the same time, scale up the accuracy with the remaining "idle" computational resources. In addition, the locations of the maxima are resolved to a point at which the total amount of double-precision calculations is small enough that, given even a minimal number of CPU cores, these calculations can be run on the CPU in a similar time to the GPU computation.

As an alternative, the performance of the optimisation can be measured using the metrics of Section 6.1.2. In the case examined, if the time of the speed metric is taken as the asynchronous run time of the CO stage using the default configuration, and the total number of COPs performed in this time is calculated, an average of 352.27 CCpCOP is obtained. This real-world use is a mix

of both the `pln_nrw` and `pln_brd` kernels. Thus, the observed value of 352.27 may seem somewhat high when compared to the theoretical worst case of ∼240 CCpCOP. However, the values of Section 6.1.2 were obtained in optimal conditions. In real-world use cases, many other factors come into play which may decrease performance; the most notable of these is that the GPU computation may not be the limiting factor in the above case. However, in our experience, the speed of the CO stage will generally range from 280 to 350 CCpCOP. On the hardware used in our testing, this translates to between 200 and 600 candidates optimised per second in a search summing up to 16 harmonics, and between 2200 and 3600 candidates optimised per second in a search summing only 1 harmonic.

7.4.8 Summary of configuration parameters

We designed `AccelGPU` such that if the configuration parameters are left at their default values, the application will select reasonable values that give close-to-optimal search speeds. However, tuning these parameters by hand can often give slight improvements to the speed of a search. Here, we give a brief summary of the primary configuration parameters and their effects.

When summing more than two harmonics, an in-memory search should be used wherever possible, as this has the greatest impact on speed. Second to this, the plane width has one of the largest effects for most searches, a plane width of either 4092 or 8192 is optimal, with 2048 occasionally being optimal when summing lower numbers of harmonics or when using older GPUs. We recommend using an initial plane width of 4092 and keeping the amount of contamination in the primary plane below 20%. The amount of work assigned to each processing unit during the CG stage can be balanced by changing which of the two units performs the normalisation and FFT components. When $Z_{\max}$ is small, more work should be performed by the GPU, whereas when $Z_{\max}$ is large, the run time can often be reduced by shifting more of the work to the CPU. The N_s parameter allows some control over the granularity of work done in the CG stage. Generally, larger values of N_s improve the performance, especially on newer GPUs with more registers, although care must be taken not to increase N_s to the point at which register spilling occurs. The amount of high-level concurrency is controlled with the N_g and N_o parameters; on faster CPUs or computers with many cores, increasing these parameters may improve performance. However, beyond some point, this can increase GPU contention, which can decrease performance. The N_s

and N_g parameters determine how much GPU memory will be used, care should be taken not to use so much memory that an in-memory search is not possible.

The results shown in this chapter demonstrate that the FDAS has been successfully accelerated. The largest portion of this acceleration has been achieved by porting performance-critical components to the GPU. However, in order to see the scale of these speed-ups reflected in the run time of the full search, this requires a highly optimised and tuned asynchronous pipeline. Our pipeline is highly configurable, and our extensive profiling shows the effects of, and interplay between, these configurations. When these are combined effectively, the run time of the search can be significantly reduced.

Chapter 8

Conclusions

Accurate measurement of pulsar spin frequency is an exemplary measurement tool that allows physicists to accurately probe a range of physical conditions that are far removed from those in our solar system. The value derived from observing pulsars means that there is an ongoing drive to discover new ones, especially exotic pulsars in relativistic orbits. However, finding these very faint stellar objects is hindered by the long integration times and very large data sets that are needed to detect them. The primary aim of this work is to analyse the FDAS in order for GPU implementations to be designed and written in the best way possible - both ours and future implementations. This has the intention of reducing the time and cost of performing an FDAS to the point at which the acceleration searches of future large-scale surveys could be performed in real time. To demonstrate this, we developed a GPU-accelerated implementation of the FDAS that runs 30 to 70 times faster than the existing serial CPU implementation, using a single desktop GPU.

8.1 Summary of work

Optimising this application entailed a cyclical process of analysis, design, implementation, and testing. We initially performed a numerical analysis of the key computations of the FDAS, given that such an analysis had not previously been published. This allowed us to identify the computations that caused the greatest errors in the coefficients of the various filters used in this method. The largest of these errors was found in the calculation of Fresnel integrals. We developed a highly efficient routine that exploits the IEEE floating-point representation to increase the numerical accuracy of the calculation of the Fresnel integrals at a given precision. This enabled us to develop a fast and efficient function to calculate acceleration coefficients in both single- and double-precision, on the GPU and the CPU. We identified that despite this increased accuracy, there are regions where the error is nonetheless significant; we found this to be particularly true

at very low accelerations. To mitigate the impact of the error in this range, a new approximation was developed that increases the accuracy of the low-acceleration coefficients. We developed a number of GPU-accelerated host functions which are capable of calculating any of the relevant coefficients as well as sections of r-$\dot{r}$ plane. An evaluation of the numerical accuracy and speed of the coefficient calculations is presented as part of this work.

We profiled `Accelsearch`, the existing CPU implementation of the FDAS, and established that it comprised two stages: candidate generation (CG) and candidate optimisation (CO). We developed a GPU-accelerated port of this implementation; this is designed as a flexible search tool and is not tailored for any specialised use, such as a specific survey. The implementation is highly configurable and is designed for use on multiple GPUs, which is a core objective of this work. Analysis of the pipeline identified that there are multiple levels at which parallelism can be exploited in the CG as well as the CO stage, both of which are iterative in nature. The low granularity of these iterations is significant in that it impedes some opportunities for optimisation; this necessitated some attentive design to create a highly asynchronous search pipeline.

Analysis of the CG stage revealed the potential for a number of beneficial modifications. Given that GPUs are generally considered to be a memory-limited environment, two related changes were made: the first was splitting the in-memory r-$\dot{r}$ plane in two, the second was storing the powers in half-precision. The combination of these enhancements quadruples the size of the data that can be processed using the faster in-memory variant of the search on a given GPU. To fully utilise the remaining device memory, a new scalable plan data structure was created. This scalability was implemented in such a way that units of work can now be aggregated, allowing increased device occupancy. In addition, it was identified that plane width plays an important role in determining the minimum run time; we moved from a hardcoded plane width, which limits flexibility, to a dynamic plane width. This minor alteration has the potential to – in some cases – halve the run time of the existing CPU search. Using these changes, each component of the stage was implemented and iteratively optimised. These were combined into a fully asynchronous pipeline which makes use of multiple CPU threads, asynchronous CUDA kernels, and asynchronous memory copies.

In the CO stage, the r-$\dot{r}$ location of each candidate is refined; this refinement is a 2D, derivative-free optimisation and is the only substantial component of this stage. The existing location re-

finement makes use of the Nelder–Mead method, which is not well suited to parallelisation, and therefore not to GPU computation either. Our focus was thus on integrating a new derivative-free optimisation, better matched with the GPU paradigm. To this end, an iterative pattern search was selected; this new method has the advantage that it evaluates a significantly wider range of the r-$\dot{r}$ space, but this comes at the cost of greater computation. The computational power of the GPU can thus be leveraged to increase both the speed and the accuracy of the CO stage. Subsequent to implementing this new method, testing and analysis were undertaken to quantitatively evaluate the performance of both stages of the search.

8.2 Discoveries

The primary objective of reducing the run time of the FDAS has been successfully achieved with our GPU-accelerated implementation. Our tests demonstrate that the run time of a single DM search can be reduced by 30 to 70 times, with a single desktop GPU from the Pascal generation. These speed-ups are consistent with industry standards. Furthermore, the speeds achieved allow 1100 to 2200 SKA-equivalent DM-corrected time series to be searched within the duration of the observation, when up to 16 harmonics are summed and 100 acceleration trials are examined. Thus, the 500 DM trials across all 1500 beams of the SKA1-mid survey could be searched in real time with 340 to 675 desktop GPUs. This brings the GPU-accelerated FDAS into the range of real-time processing, even for the proposed SKA surveys.

The CG stage represents the majority of the computation load of the search, this stage was therefore the main target of our optimisation. This resulted in a search rate of between 1600 and 40000 for this stage, which drastically reduced overall run times. It was found that once all the components of the stage had been optimised, the run time of the CG stage of a typical search was solely determined by three primary components: multiplication, iFFT and SAS. These three components are run on the GPU. Due to the scaling of these components, the run time of the CG stage scales linearly with observation length, the number of acceleration trials, and the number of harmonics summed. The speed-up and the scaling shown here will translate to other applications, since the computation in the CG stage is fairly predictable.

We find that in the faster in-memory variant of the CG stage, the most significant component is harmonic summing and searching, accounting for approximately 60% of run time. The standard

variant of the search is dominated by the iFFT component which accounts for 46% of the run time. The primary configuration parameter of the CG stage is plane width. A large plane width will significantly increase the run time of the iFFT component, while too small a plane width will increase the contamination, and thus increase "superfluous" computation. We find that a plane width of 4096 generally performs best, although the optimal plane width is dependent on many factors, including the hardware and the CUDA runtime version – and thus cuFFT library version – used. In this work, the largest performance gains were achieved by porting the computation of individual components to the massively parallel GPU architecture. However, this works in conjunction with our highly tuned asynchronous pipeline. This pipeline allows us to exploit multiple levels of both data and task parallelism, meaning that we can fully saturate a GPU with work. Thus, for reasonably sized searches, the factors limiting the run time of the CG stage are the bandwidth and the computational power of the GPU.

The new candidate optimisation method not only decreases run time but simultaneously increases accuracy as well. This method is highly reliant on calculating many acceleration coefficients at arbitrary locations. Accordingly, we optimised the calculation of these coefficients and constrained the maximum error to 1e-5 and 2e-12 for single- and double-precision respectively. Using our GPU-accelerated coefficient calculation function, it is possible to calculate a single-precision acceleration filter coefficient in approximately 230 GPU clock cycles. This enables us to calculate up to 1000 single-precision r-$\dot{r}$ points per microsecond using our `rr_plane` function. This, in turn, allows the new pattern search to calculate orders of magnitude more r-$\dot{r}$ points than the existing implementation, exploring a wider range of the parameter space and increasing the likelihood of identifying the location of the true maximum. This, however, limits the speed-up of the CO stage to only 10 to 30 times faster, which nonetheless allows us to optimise up to 500 candidates per second when summing 16 harmonics. The increase in speed is not as great as in the CG stage, however, the increased accuracy correctly identifies spurious duplicate candidates. This is a nontrivial improvement as for a human operator to investigate a spurious candidate requires significantly more time than running an entire search of a single DM trial, on the GPU.

8.3 Limitations

While the goal of this work was achieved, there are some limitations to the research and its scope. One of these is that the search was only tested and optimised on a limited range of hardware; thus the results may not transfer to the most recent generations of GPUs, as well as to compute-specific GPUs. Moreover, these results may lose relevance with rapid innovations in GPU technology. The search was not tested in a real-world search pipeline, as the researchers were not associated with an active pulsar research group. The implementation may have benefited from testing in a number of typical use cases. Furthermore, the parameter tuning of the new optimisation technique was not extensively tested with a very wide range of candidates: neither a large number of real nor spurious candidates. It would likely have benefited from parameter tuning across a wider range of typical observations.

Through our research, we became aware of a number of points at which the CPU version could be optimised. We did not implement these optimisations, thus our research is limited by not having been compared to the optimal CPU implementation. A similar limitation is that this work was not compared to the parallel CPU version of `Accelsearch`, which was created simultaneously to our research by S. Ransom of NRAO. While it would be expected that the performance would scale with the number of CPU cores, this has not been established as a certainty. We acknowledge that this may detract somewhat from the relevance of the speed-ups reported, however, it does not diminish the value of the absolute speeds attained through our GPU acceleration.

8.4 Opportunities for future study

This work examined a specific section of the search pipeline, whereas future work could explore the possibility of accelerating other elements of the wider pulsar search pipeline, such as folding. The parallelisation of this work could be extended to cover a multi-DM search, and could easily be expanded across multiple computational nodes. This would give the opportunity to amortise some of the computation, improving overall performance. Furthermore, it is noted that many of the GPU-accelerated components of the FDAS are memory-bound computations. Accordingly, there could be a great benefit in performing these components as pre- and post-callbacks to the iFFT. We have investigated this, and did not find the expected performance gains, however, these

could be enabled by future versions of the cuFFT library. Similarly, a hand-tuned iFFT incorporating the multiplication component, such as that used by Dimoudi et al., could negate the run time of the multiplication component. During the course of our work, we noted that sections of the correlation kernel were very close to zero. Further research could examine whether incorporating this into a hand-tuned iFFT kernel could improve performance further, by skipping certain computations in the multiplication and iFFT components. Another opportunity for future research centres around developing an auto-tune feature for our implementation which could establish the optimal parameters for a specific hardware environment. This is made possible by the high degree of configurability of our search, and identifying optimal combinations of configuration parameters on a specific hardware could have a significant impact on performance. Lastly, in our work, we have identified a number of areas where the existing CPU implementation could benefit from optimisation and parallelisation across multiple CPU cores; future research could focus on implementing these improvements in the CPU version.

Our results were obtained using desktop GPUs. However, future large-scale pulsar surveys would likely make use of compute-specific GPUs. These GPUs usually cost more per-FLOPs, however, they are more reliable, have better double precision performance, and increased device memory. Furthermore, compute-specific GPUs usually have significantly higher bandwidth than contemporary desktop GPUs, and it may be expected that future generations of compute-specific GPUs will continue this trend of having higher bandwidth. As we demonstrated, the current implementation of the CG stage of the FDAS is bandwidth limited. Thus, there is scope to tune our GPU-accelerated FDAS to take advantage of these high bandwidth GPUs.

This work represents not only a proof of concept but a working application; the hope is that it will be integrated into existing or future research projects. The FDAS is well suited to being performed in parallel and the lack of a previous parallel implementation has no doubt hindered advancement in the field. We crafted a highly optimised, fully functional open-source implementation which has many potential uses, though these have not yet been widely tested. As an example, though this implementation was not designed for the SKA, it would allow real-time searching of SKA1-mid survey data using 340 to 675 desktop GPUs. This work produced a faster and more cost-efficient search capability, and as an additional output, has developed a more accurate method of computing Fresnel integrals. The principle contribution of this work to the field of pulsar

astronomy is that it offers both a set of relevant results and also a multi-purpose tool that may be used by the wider research community.

Acronyms

CCpCOP

clock cycles per convolution operation. 134–136, 194, 233, 234

COP

convolution operation. 133, 134, 194, 233, 245, 258, *Glossary:* convolution operation

CUDA

Compute Unified Device Architecture. 4, 7, 57–61, 64–74, 76, 81, 82, 84, 86, 93, 95–99, 101, 104, 105, 108–115, 131, 133–135, 140–145, 148, 152, 157–160, 162, 163, 178, 179, 186, 200, 205, 211, 214, 215, 218, 222, 225–227, 232, 238, 240, 247, 261, 265, *Glossary:* Compute Unified Device Architecture

DM

dispersion measure. 18, 19, 24, 27–30, 35, 78, 80, 84–86, 180, 181, 191, 233, 239–241, *Glossary:* dispersion measure

ISM

interstellar medium. 2, 18, 259, *Glossary:* interstellar medium

MSP

millisecond pulsar. 1, 11–15, 19, 29, 37, *Glossary:* millisecond pulsar

PTA

pulsar timing array. 1, 11, 12, 14, *Glossary:* pulsar timing array

SAS

sum-and-search. 105, 106, 150–152, 154–156, 158, 159, 162, 180, 185, 187, 192, 203, 206–214, 216, 218, 220, 223, 226–230, 239, 253, 258, *Glossary:* sum-and-search

SM

streaming multiprocessor. 67–72, 74–76, 114, 218, 226, 227, 232, 261, 264, 265, 284, *Glossary:* streaming multiprocessor

TOA

time of arrival. 2, 11, 12, 14, *Glossary:* time of arrival

2D

two-dimensional. 4, 33, 35–38, 44, 46, 55, 57, 61, 63, 81, 98, 108, 130, 131, 141, 238

API

application programming interface. 4, 7, 57, 59, 65, 72, 74, 109, 128, 258, 262

CC

clock cycles. 76, 115–117, 119–122, 125, 129, 130, 284

CCpC

clock cycles per coefficient. 129, 134, *Glossary:* CCpC

CCpO

clock cycles per operation. 113, 114, 285

CG

candidate generation. 40, 42, 46–48, 54, 87–98, 101–103, 105, 108, 110, 111, 113, 130, 133, 136–141, 147, 148, 151, 153, 155–158, 160–165, 168, 170, 180, 182, 183, 185–187, 189–195, 197–200, 202, 203, 207–217, 219–231, 233, 234, 238–240, 242, 253, 264

CO

candidate optimisation. 40, 54, 80, 84, 87–92, 106, 109–111, 130, 161–163, 165, 180, 183, 186, 187, 189–193, 196–200, 207–210, 213–215, 230–234, 238–240, 253, 258, 261

CPU

central processing unit. i, 4–7, 28, 30, 38, 39, 57–59, 73, 74, 77, 79, 82, 83, 85, 87, 89, 90, 96–98, 101, 103, 104, 106, 109, 110, 114, 122, 128, 133, 140, 148, 157–167, 172,

177–179, 186–188, 195–200, 203–206, 210, 211, 213–215, 221–225, 228–234, 237, 238, 241, 242, 257, 260

cuFFT

CUDA Fast Fourier Transform library. 58, 64, 65, 81–85, 95, 103–105, 139, 141, 142, 145, 148, 200, 202, 228, 240, 242

D2H

device-to-host. 158–160

DFT

discrete Fourier transform. 4, 20–23, 31, 39–42, 44, 46, 47, 50, 82, 85, 89, 90, 92, 97, 102, 103, 106, 111, 112, 115, 123, 130, 131, 139, 141, 158, 161, 164, 166, 181–183, 185, 186, 196, 203, 216, 218, 253, 257, 259, 263

DIF

decimation in frequency. 82

DIT

decimation in time. 82

DPS

dynamic power spectrum. 26, 36–38

DRAM

dynamic random-access memory. 62, 63

DSP

digital signal processing. 2, 3, 5, 15, 78

ECC

error-correcting code. 62

EM

electromagnetic. 1–3, 10, 17, 18

FAST

Five-hundred-meter Aperture Spherical Telescope. 3

FDAS

frequency domain acceleration search. i, 4–7, 16, 17, 27, 29, 30, 33, 35, 39–42, 44, 46, 48, 50, 52, 54, 80, 82, 84–88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110–112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 177, 180, 191, 200, 235, 237–239, 241, 242, 257, 258, 261, 263, 264, *See: Section 2.3.4.2*

FFA

fast folding algorithm. 19

FFT

fast Fourier transform. 4, 21, 22, 36, 43, 44, 49, 50, 64–66, 78, 81–85, 88, 89, 97, 103, 104, 140, 141, 148, 158, 180, 181, 185, 192, 203, 205, 206, 210, 211, 214, 218, 221, 222, 234

FFTW

The Fastest Fourier Transform in the West. 64, 85, 95, 104, *Glossary:* FFTW

FIR

finite impulse response. 23, 30, 31, 115, 117

FLOPs

floating-point operations per second. 70, 71, 76, 77, 242

FMA

fused multiply-add. 119

FPA

focal plane array. 17

FPGA

field-programmable gate array. 80

FT

Fourier transform. 21, 23, 30, 35, 36, 43, 44, 81, 141, 181, 263

GBT

Green Bank Telescope. 181

GFLOPs

giga-floating-point operations per second. 66

GPGPU

general-purpose graphics processing unit. 7, 57, 58, 64, 66, *See Section 4.1*

GPU

graphics processing unit. i, 1, 4–7, 17, 28, 35, 38, 39, 57–59, 62–74, 76–86, 88–120, 122, 124, 126, 128–136, 138, 140–142, 144, 146–150, 152, 154–168, 170–174, 177–180, 182, 184, 186–188, 190–192, 194–196, 198, 200, 202–206, 208–218, 220–226, 228–235, 237–242, 257, 258, 264, 265, 284, *Glossary:* graphics processing unit

GR

general relativity. 1

H2D

host-to-device. 104, 158–160, 203, 206, 211, 213, 220

HPC

high-performance computing. 1, 4, 57, 58

HTRU

High Time Resolution Universe. 29

I/O

input/output. 58

iDFT

inverse discrete Fourier transform. 44

IEEE

Institute of Electrical and Electronics Engineers. 71, 72, 237

iFFT

inverse fast Fourier transform. 43, 49, 50, 81, 89, 91, 99, 104, 105, 145, 147, 148, 150, 158, 185, 192, 200, 202, 203, 206, 208–212, 214, 216–218, 220, 222, 223, 228, 239–242

iFT

inverse Fourier transform. 43

ILP

instruction-level parallelism. 69, 72, 75, *Glossary:* instruction-level parallelism

NRAO

National Radio Astronomy Observatory. 241

OpenCL

Open Compute Language. 58

OS

operating system. 57

PC

personal computer. 66, 77

PMPS

Parkes multibeam pulsar survey. 17, 29

POSIX

Portable Operating System Interface. 157, 161, 162, 164

PTX

parallel thread execution. 122, *Glossary:* PTX

RAM

random-access memory. 62, 64, 66

RFI

radio frequency interference. 50, 87, 88, 161, 163, 167, 181, 182, 187, 191, 192, 207, 210, 215

ROI

region of interest. 124–129

SFU

special function unit. 67, 71–73, 114, 115, 264

SKA

Square Kilometre Array. i, 3, 28, 77, 80, 83, 84, 181, 182, 188, 191, 239, 242

SNR

signal-to-noise ratio. 11, 23, 25

SP

stream processor. 68

TDAS

time domain acceleration search. i, 27–30, 35, *See: Section 2.3.4.2*

Symbols

N_Z Number of acceleration trials. 39, 47, 81, 83, 84, 212

N_c Number of elements in a SAS chunk. 153, 155, 156, *See: Section 6.2.3.6*

N_g Number of concurrent threads used in the main loops of the CG stage. 96, 186, 187, 191, 202, 224–226, 234, 235

N_o Number of concurrent threads used in the main loops of the CO stage. 110, 163, 186, 187, 234

N_s Number of segments aggregated together into a batch. 96, 101, 141, 144, 145, 147, 149, 152, 155, 156, 186, 187, 191, 202, 223–226, 234

N_t The number of samples in an observation. 20–22, 78, 83, 182, 183, 189, 191, 194, 195, 197, 207, 208

P_{orb} The period of an orbit. 26, 28, 36, 37

S_w Segment size, this is the width of the DFT values used to form the primary plane. 44, 46, 47, 51, 91, 98, 188, 216, 263

T_{obs} The duration of an observation. 20, 22, 26, 28, 29, 31, 36, 39, 254

$Z_{\max}$ The $Z_{\max}$ indicates the extent of the acceleration trials explored by `Accelsearch`. It is measured in $\dot{r}$, the number of spectral bins the signal drifts by during an observation. It dictates the height of the primary plane, each stage of harmonic summing will search between $\pm$ the harmonic fraction of $Z_{\max}$. 39, 43, 44, 46, 87, 88, 91, 92, 95, 97, 99, 137, 139, 147, 155, 156, 179, 181–184, 186–189, 192–195, 197, 199, 200, 202, 207, 209, 210, 213–216, 218–224, 226, 228–230, 234, 254, *See: Chapter 3*

Z The number of spectral bins a signal changes by during an observation ($\dot{r}$). 39, 92, 182–184, 192, 226, *See: Chapter 3*

Δt The time separation of samples in a time domain signal. 20, 84

$\dot{r}$ The number of spectral bins the frequency changes by during the observation, $\dot{r} = \dot{f}{T_{\text{obs}}}^2$. 31–34, 39, 40, 43, 44, 46–51, 92, 112, 113, 123–129, 131–134, 145, 151, 152, 157, 168, 254, 261, *See: Chapter 3*

ω Plane width, this is the width of the primary plane measured in number of values. 43, 44, 46, 47, 49, 50, 82, 83, 91, 95, 101, 137, 147, 155, 156, 179, 184–187, 205, 210, 212, 216, 217, 219, 224, 226, 262, *See: Section 3.3.1*

a The line-of-sight acceleration of a pulsar. 28, 39

f_{spin} Pulsar spin frequency. 21, 24, 27, 28, 36, 37, 39, 46

h_s Number of harmonics in a stack. 101, 147, *See: Section 5.3.2*

h The number of harmonics summed. 24, 27, 28, 39, 46, 93, 133, 134, 145, 153, 155, 156, 184, 185, 187–189, 194–197, 200, 205, 207, 211, 213, 216, 220, 224, 225, 228

k Spectral bin index, this is a whole number which indicates the Fourier frequency. *See: Section 2.3.3.1*

m The width of an acceleration filter. 23, 30, 33, 43, 44, 47, 49

r Spectral bin index, this is a real value which indicates the Fourier frequency. 22, 23, 31, 33, 34, 39, 40, 46, 47, 50–52, 54, 82, 98, 106, 131, 132, 151, 152, 157, 183, 200–202, 261, *See: Section 2.3.3.1*

t The half-width, half of the width of the widest acceleration filter used in the convolution kernel of a r-$\dot{r}$ plane. 44, 91, 260, *Glossary:* half-width

Glossary

`AccelGPU`

The new GPU-accelerated implementation of the FDAS develops in this work.. 7, 39, 85, 86, 91–93, 97, 101, 102, 110, 118, 124, 141, 177, 181, 184, 186, 196, 198, 200–202, 224, 229, 233, 234

`Accelsearch`

The existing serial CPU implementation of the FDAS. This application is part of the greater pulsar search suite PRESTO. i, 5, 7, 32, 39–41, 43, 47–49, 51, 54, 80–83, 85, 86, 88, 90–93, 101, 102, 106, 111, 118, 122, 124, 151, 157, 177–184, 195–197, 200, 238, 241, 254, 263

backend

The custom hardware and software used to process the data from a telescope. 2, 6, 17, 18, 58, 80

batch

A batch is a collection of multiple families of r-$\dot{r}$ planes, with each family being related to a separate segment of the input DFT. The memory needed to store the various planes of a batch is allocated as a single contiguous block of memory, which is subdivided into stacks and planes. 95–101, 106, 137, 138, 141, 153, 154, 158–161, 183, 186, 192, 207, 218, 223, 224, 253, 262, *See: Section 5.3*

bus

In computer architecture, a bus is a communication system that transfers data between components inside a computer, or between computers. This expression covers all related hardware components and software, including communication protocols. 6, 64, 66, 75, 96, 97

candidate storage

A component of the FDAS, which takes the candidates found by the SAS component, calculates their sigma values and stores them for later processing in the CO stage. 89, 97, 106, 153, 157, 158, 161, 162, 180, 187, 197, 211, 229, 230, *See: Section 5.3.3.11*

CCpC

The number of GPU clock cycles for a single GPU core to calculate a single filter coefficient, averaged over a large number of threads and operations. 129, *See: Section 6.1.1.4*

CCpCOP

The number of GPU clock cycles for a single GPU core to perform a single COP, averaged over a large number of threads and operations. 134–136, 194, 233, 234, 245, *See: Section 6.1.1.4*

CCpO

The number of GPU clock cycles for a single FP32 core to perform a given calculation, averaged over a large number of threads and operations. 113, *See: Section 6.1.1.1*

compute capability

Compute capability is a versioning number used on NVIDIA compute capable hardware. 59, 62, 69, 71, 73, *See: Section 4.3*

Compute Unified Device Architecture

A parallel computing platform and API model created by Nvidia for general-purpose GPU computation. 4, 7, 57–61, 64–74, 76, 81, 82, 84, 86, 93, 95–99, 101, 104, 105, 108–115, 131, 133–135, 140–145, 148, 152, 157–160, 162, 163, 178, 179, 186, 200, 205, 211, 214, 215, 218, 222, 225–227, 232, 238, 240, 245, 247, 261, 265

configuration parameters

The parameters that determine how a search is performed, these parameters for the most part do not affect the range or resolution of the pulsar-specific parameters that are searched. The range and resolution are set by the search parameters. 8, 46, 97, 101, 150, 156, 167, 177, 185, 188, 189, 202, 215–218, 224, 242, *See: Section 3.3*

convolution kernel

A convolution kernel is the Fourier transform of a function or filter. It is used to perform a convolution using the convolution theorem. In this text, the term is predominantly used to refer to the collection of convolution kernels required to generate a section of r-$\dot{r}$ plane. 43, 44, 46, *See: Section 3.2.1*

convolution operation

The basic operation used in performing a manual convolution (Equation 2.9). At the most basic level, this comprises the possible calculation and multiplication of a filter coefficient with some DFT bin value. 133, 134, 194, 233, 245, 258, *See: Section 6.1.2*

dedispersion

Removing the frequency-dependent dispersion caused by the interactions of electromagnetic radiation with ionised particles in the ISM. 2, 16, 18, 19, 78, 180, *See: Section 2.3.2*

die

The die is a rectangular pattern on a silicon wafer that contains circuitry to perform a specific function. 68

dispersion

Dispersion is the phenomenon in which the phase velocity of a wave depends on its frequency. 2, 3, 15, 16, 18, 25, 78, *See: Section 2.3.2*

dispersion measure

Dispersion measure is the integrated column density of free electrons between an observer and a pulsar. 18, 19, 24, 27–30, 35, 78, 80, 84–86, 180, 181, 191, 233, 239–241, 245

duty cycle

A duty cycle is the fraction of one period in which a signal is active. In pulsars, this describes how narrow the pulse profile is, and is given as pulse width over spin period. 23–25, 50, 195, *See: Section 2.3.3.3*

epsilon value

The smallest value a floating-point value can differ from zero by and is given as 2^a where a is the number of bits used to store the floating-point mantissa. 93

family

In this text, family is used to refer to a collection of harmonically related items, particularly a collection of r-$\dot{r}$ planes, including the fundamental and all harmonics. 51–53, 95, 98–102, 137, 151, 153, 184, 185, 194, 206, 212, 257, 262, *See: Section 3.4.3*

FFTW

The Fastest Fourier Transform in the West is a highly optimised and widely used software library for computing discrete Fourier transforms on the CPU. 64

folding

The process of dividing a pulsar observation covering may pulses, into short sections each containing a single plus and summing these together. This forms an integrated pulse profile. 11, 38, 80, 241

graphics processing unit

A graphics processing unit (GPU) is a single-chip processor containing many computational cores and a number of memories. Traditionally, GPUs are used to create lighting effects and transform objects in a graphical 3D Environment, however there is a growing trend to use GPUs for general-purpose computation. i, viii–xi, 1, 4–7, 17, 28, 35, 38, 39, 57–59, 62–74, 76–86, 88–120, 122, 124, 126, 128–136, 138, 140–142, 144, 146–150, 152, 154–168, 170–174, 177–180, 182, 184, 186–188, 190–192, 194–196, 198, 200, 202–206, 208–218, 220–226, 228–235, 237–242, 249, 257, 258, 264, 265, 284

half-width

Half the width of the largest filter used to create a section of r-$\dot{r}$ values, this represents the maximum width of wraparound contamination caused by the correlation method. This is shown as t in Figure 3.3. 44, 46, 137, 255, *See: Section 3.2.2*

instruction-level parallelism

Instruction-level parallelism, is a measure of how many of the instructions in a computer program can be executed simultaneously. 69

interbinning

A method for approximating Fourier amplitudes at half-integer frequencies using only the two neighbouring integer-frequency bins. 22, 23, 47, *See: Section 2.3.3.2*

interstellar medium

The interstellar medium is the matter and radiation that exists in the space between the star systems in a galaxy. 2, 18, 245, 259

intrinsic

Intrinsics are functions that are built into the compiler. In CUDA, some of these intrinsic functions map directly to hardware operations.. 4, 60, 62, 72, 73, 76, 82, 116

location refinement

A component of the CO stage of the FDAS. This component refines the r and $\dot{r}$ locations of the initial candidates. 55, 91, 92, 107–111, 130, 163–169, 171–174, 210, 214, 215, 230–232, *See: Section 3.4.6*

lockstep

Lockstep describes the execution model of an SM, in which all threads of a warp are simultaneously issued the same instruction, and all instructions are performed in unison by all the threads of the warp. 60, 68, *See: Section 4.2.1*

magnetar

A magnetar is a type of neutron star believed to have an extremely powerful magnetic field. The magnetic-field-decay powers the emission of high-energy electromagnetic radiation, particularly X-rays and gamma rays. 12

millisecond pulsar

A millisecond pulsar is a pulsar with a rotational period in the range of about 1-10 milliseconds, formally we use the empirical definition $\frac{\dot{P}}{10^{-17}}\left(\frac{P}{100ms}\right)^{2.34} \leq 2.23$ defined by Lee et al. [64] . vii, 1, 11–15, 19, 29, 37, 245

multi-segment stack

A collection of planes from a number of segments all of the same width, usually allocated as an aligned and strided block of memory. *Glossary:* segment & stack

OpenMP

OpenMP is an API that enables parallel programming on many platforms and a range of development languages.. 82, 148, 158, 163

pinned

Pinned memory is memory allocated by the `cudaMallocHost` function. This memory is non-pageable, meaning that it cannot be paged out to virtual memory, which can decrease transfer speeds. 64, 103, 106, 139, 161, 165, *See: Section 4.2.2.7*

plan

A custom data structure that encapsulates the run-time configuration, execution plan, and data required to create a batch of r-$\dot{r}$ planes. x, xi, 95–98, 101–103, 106, 110, 137–140, 147, 148, 150, 151, 155–159, 162, 163, 165, 186, 187, 191, 202, 203, 214, 222–224, 226, 227, 230, 238, *See: Section 5.3*

plane width

The full width of a section of the r-$\dot{r}$ plane, this includes the sections contaminated by the wraparound effects of the convolution kernel. This is shown as ω in Figure 3.3. 46, 97, 101, 137, 151, 179, 186, 188, 196, 203, 205, 216–220, 222, 226, 227, 234, 238, 240, 254

primary plane

The largest plane in a family of r-$\dot{r}$ planes. 46, 51–53, 98, 181, 184–186, 206, 218–220, 253, 254, *See: Section 3.4.3*

PTX

Parallel thread execution, is a low-level parallel thread execution virtual machine and instruction set architecture. 122

pulsar timing array

A collection of pulsars, ideally isotropically distributed over the whole sky, which are accurately and regularly timed for a long period of time, with the hope of measuring gravitational waves. 1, 11, 12, 14, 245

radial velocity

The relative velocity of a body along the line of sight of an observer. 27–29

search parameters

The *a-priori* unknown parameters that describe the properties of the signal from a pulsar. In a search, trial values of these parameters are iterated over. 6, 15, 16, 18, 24, 39, 40, 46, 85, 91, 177, 178, 183, 188, 207, 216, 224, 258, *See: Section 2.3*

search rate

Search rate is a speed metric that is distinct from the standard speed. Search rate is the number of a specific task that can be performed in the duration of a pulsar search observation. 180–182, 187–189, 191, 224, 225, 239, *See: Section 7.1.2*

segment

A small section of the input DFT searched by the FDAS. These small sections of the input data are what are iterated over during the main search loop in `Accelsearch`. xi, 40, 42, 47, 53, 95, 96, 100, 101, 141, 144, 146, 147, 149, 151–156, 186, 192, 205, 216, 220, 222–224, 227, 253, 257, 262, 263, *See: Section 3.1*

segment size

The width of the segments, in Fourier Bins, into which the input FT is divided. This dictates the size of the "used" part of an r-$\dot{r}$ kernel, shown as S_w in Figure 3.3. 46, 91, 98, 137, 178, 179, 186, 220, 253, *See: Section 3.4.2*

stack

A collection of planes, all of the same width, usually allocated as an aligned and strided block of memory as shown in Figure 5.4. 98, 99, 101–103, 105, 137–139, 141, 142, 144, 147–149, 184–186, 226, 227, 254, 257

streaming multiprocessor

A streaming multiprocessor is a physical part of a GPU architecture. Each streaming multiprocessor contains thousands of registers, many computation units, and various caches. 67–72, 74–76, 114, 218, 226, 227, 232, 246, 261, 264, 265, 284

strided

Strided refers to a manner of storing data in memory, whereby data is stored in a number of non-contiguous memory locations spaced by a constant amount of memory known as the "stride". 131, 138, 140

subpartition

An SM is divided into subpartition(s). Each subpartition contains an instruction scheduler, a number of FP32 and FP64 cores, an SFU and a number of 32-bit registers. Each warp resides on a single subpartition. 67–70, 73, 76, *See: Section 4.3*

sum-and-search

Sum-and-search is one of the components of the GPU implementation of the CG stage of the FDAS. 105, 106, 150–152, 154–156, 158, 159, 162, 180, 185, 187, 192, 203, 206–214, 216, 218, 220, 223, 226–230, 239, 245, 253, 258, *See: Section 5.3.3.9*

time of arrival

The arrival time of some fiducial point on the integrated profile with respect to either the start or the midpoint of an observation. 2, 11, 12, 14, 246

transcendental

A transcendental number is one that is not a root of any algebraic equation having integral coefficients, as π or e. Similarly a transcendental function is an analytic function that does not satisfy a polynomial equation, this is in contrast to an algebraic function. Examples

of common transcendental functions are trigonometric functions, reciprocals, square roots, logarithms, and many graphics interpolation instructions. 65, 71, 72, 113

warp

A warp is a collection of 32 CUDA threads. All the threads in a warp execute instructions concurrently on the resources of the SM. 60–63, 66–70, 72–76, 130, 261, 264, *See: Section 4.2.1*

word

In computing, a word is the natural unit of data used by a particular processor design. Most modern GPUs have a word size of 32 or 64 bits. 67, 72

Bibliography

[1] Abbott B. P. Abbott R. Abbott T. D. Abernathy M. R. Acernese F. et al. 2016. Observation of gravitational waves from a binary black hole merger. *Phys. Rev. Lett.* 116, 6 (2016), 1–16. DOI:http://dx.doi.org/10.1103/PhysRevLett.116.061102

[2] Anderson B. Lyne A. G. 1983. On the origin of pulsar velocities. *Nature* 303, 5918 (jun 1983), 597–599. DOI:http://dx.doi.org/10.1038/303597a0

[3] Antoniadis J. Freire P. C. C. Wex N. Tauris T. M. Lynch R. S. et al. 2013. A Massive Pulsar in a Compact Relativistic Binary. *Science (80-.).* 340, 6131 (apr 2013), 1233232. DOI: http://dx.doi.org/10.1126/science.1233232

[4] Armour W. Karastergiou A. Giles M. Williams C. Magro A. et al. 2011. A GPU-based survey for millisecond radio transients using ARTEMIS. 461 (2011), 33–36. http://arxiv.org/abs/1111.6399

[5] Aulbert C. 2005. *Finding Millisecond Binary Pulsars in 47 Tucanae by Applying the Hough Transformation to Radio Data*. Ph.D. Dissertation. Max Planck Institute for Gravitational Physics, University of Potsdam, Germany. http://carsten.welcomes-you.com/science/publications/phd-2006.pdf

[6] Aulbert C. 2006. Finding binary millisecond pulsars with the Hough transform. In *Proc. 363. WE-Heraeus Semin. Neutron Stars Pulsars*, Becker W. and Huang H. H. (Eds.). Max-Planck-Institut fur Extraterrestrische Physik, Physikzentrum Bad Honnef, Germany, 216–218. http://arxiv.org/abs/astro-ph/0701097

[7] Baade W. Zwicky F. 1934. Cosmic Rays from Super-Novae. *Proc. Natl. Acad. Sci.* 20, 5 (1934), 259–263. DOI:http://dx.doi.org/10.1073/pnas.20.5.259

[8] Backer D. C. Hellings R. 1986. Pulsar timing and general relativity. *Annu. Rev. Astron. Astrophys.* 24 (1986), 537–575. http://adsabs.harvard.edu/full/1986ARA

[9] Bailes M. 2009. The art of precision pulsar timing. *Proc. Int. Astron. Union* 5, S261 (apr 2009), 212–217. DOI:http://dx.doi.org/10.1017/S1743921309990421

[10] Bailes M. Barr E. Bhat N. D. Brink J. Buchner S. et al. 2016. MeerTime - the MeerKAT Key science program on pulsar timing. *Proc. Sci.* (2016). DOI:http://dx.doi.org/10.22323/1.277.0011

[11] Barr E. D. 2012. *Searching for Pulsars with the Effelsberg Telescope*. Ph.D. Dissertation. Universitäts-und Landesbibliothek Bonn.

[12] Barsdell B. R. Bailes M. Barnes D. G. Fluke C. J. 2012. Accelerating incoherent dedispersion. *Mon. Not. R. Astron. Soc.* 422, 1 (may 2012), 379–392. DOI:http://dx.doi.org/10.1111/j.1365-2966.2012.20622.x

[13] Barton P. 1985. Digital Beam Forming for Radar. *IEE Electromagn. Waves Ser.* 127, 4 (1985), 108–119.

[14] Batcher K. E. 1968. Sorting networks and their applications. In *Proc. April 30–May 2, 1968, Spring Jt. Comput. Conf.* ACM Press, New York, New York, USA, 307. DOI:http://dx.doi.org/10.1145/1468075.1468121

[15] Beliakov G. 2011. Parallel calculation of the median and order statistics on GPUs with application to robust regression. (apr 2011), 1–19. http://arxiv.org/abs/1104.2732

[16] Bhat N. D. R. Bailes M. Verbiest J. P. W. 2008. Gravitational-radiation losses from the pulsar-white-dwarf binary PSR J1141–6545. *Phys. Rev. D* 77, 12 (2008), 1–5. DOI:http://dx.doi.org/10.1103/PhysRevD.77.124017

[17] Bracewell R. 1999. *The Fourier Transform And Its Applications* (3rd ed.). McGraw-Hill Higher Education.

[18] Bregman J. D. 2004. System Optimisation Of Multi-Beam Aperture Synthesis Arrays For Survey Performance. *Exp. Astron.* 17, 1-3 (jun 2004), 365–380. DOI:http://dx.doi.org/10.1007/s10686-005-2872-8

[19] Cameron A. D. Barr E. D. Champion D. J. Kramer M. Zhu W. W. 2017. An investigation of pulsar searching techniques with the fast folding algorithm. *Mon. Not. R. Astron. Soc.* 468, 2 (jun 2017), 1994–2010. DOI:http://dx.doi.org/10.1093/mnras/stx589

[20] Camilo F. Lorimer D. R. Freire P. C. C. Lyne A. G. A. G. Manchester R. N. N. 2000. Observations of 20 millisecond pulsars in 47 Tucanae at 20 centimeters. *Astrophys. J.* 535, 2 (2000), 975. http://iopscience.iop.org/0004-637X/535/2/975

[21] Castelvecchi D. 2018. How gravitational waves could solve some of the Universe's deepest mysteries. *Nature* 556, 7700 (2018), 164–168. DOI:http://dx.doi.org/10.1038/d41586-018-04157-6

[22] Chandler A. M. 2003. *Pulsar searches: From radio to gamma-rays*. Ph.D. Dissertation. California Institute of Technology. https://resolver.caltech.edu/CaltechETD:etd-01232003-213508

[23] Chandrasekhar S. 1931. The Maximum Mass of Ideal White Dwarfs. *Astrophys. J.* 74, 2 (jul 1931), 81. DOI:http://dx.doi.org/10.1086/143324

[24] Cooley J. W. Tukey J. W. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.* 19, 90 (apr 1965), 297. DOI:http://dx.doi.org/10.2307/2003354

[25] Courbariaux M. Bengio Y. David J.-P. 2014. Training deep neural networks with low precision multiplications. In *3rd Int. Conf. Learn. Represent.* 1–10. http://arxiv.org/abs/1412.7024

[26] Davis M. M. Taylor J. H. Weisberg J. M. Backer D. C. 1985. High-precision timing observations of the millisecond pulsar PSR1937 + 21. *Nature* 315, 6020 (jun 1985), 547–550. DOI:http://dx.doi.org/10.1038/315547a0

[27] De K. Gupta Y. 2016. A real-time coherent dedispersion pipeline for the giant metrewave radio telescope. *Exp. Astron.* 41, 1-2 (2016), 67–93. DOI:http://dx.doi.org/10.1007/s10686-015-9476-8

[28] Demorest P. B. Pennucci T. Ransom S. M. Roberts M. S. E. Hessels J. W. T. 2010. A two-solar-mass neutron star measured using Shapiro delay. *Nature* 467, 7319 (oct 2010), 1081–1083. DOI:http://dx.doi.org/10.1038/nature09466

[29] Dhurandhar S. V. Vecchio A. 2001. Searching for continuous gravitational wave sources in binary systems. *Phys. Rev. D* 63, 12 (may 2001), 1–25. DOI:http://dx.doi.org/10.1103/PhysRevD.63.122001

[30] Dimoudi S. Adámek K. Thiagaraj P. Ransom S. M. Karastergiou A. et al. 2018. A GPU Implementation of the Correlation Technique for Real-time Fourier Domain Pulsar Acceleration Searches. *Astrophys. J. Suppl. Ser.* 239, 2 (2018), 28. DOI:http://dx.doi.org/10.3847/1538-4365/aabe88

[31] DuPlain R. Ransom S. M. Demorest P. Brandt P. Ford J. et al. 2008. Launching GUPPI: the Green Bank Ultimate Pulsar Processing Instrument. In *Adv. Softw. Control Astron. II*. 70191D. DOI:http://dx.doi.org/10.1117/12.790003

[32] Eatough R. P. 2009. *Searching for Relativistic Binary Pulsars in the Galactic Plane*. Ph.D. Dissertation. The University of Manchester, Manchester.

[33] Eatough R. P. Kramer M. Lyne A. G. Keith M. J. 2013. A coherent acceleration search of the Parkes multibeam pulsar survey - techniques and the discovery and timing of 16 pulsars. *Mon. Not. R. Astron. Soc.* 431, 1 (feb 2013), 292–307. DOI:http://dx.doi.org/10.1093/mnras/stt161

[34] Eatough R. P. Molkenthin N. Kramer M. Noutsos A. Keith M. J. et al. 2010. Selection of radio pulsar candidates using artificial neural networks. *Mon. Not. R. Astron. Soc.* 407, 4 (jul 2010), 2443–2450. DOI:http://dx.doi.org/10.1111/j.1365-2966.2010.17082.x

[35] Elliott D. F. 1987. *Handbook of digital signal processing: engineering applications*. Vol. 26. 26–0940–26–0940 pages. DOI:http://dx.doi.org/10.5860/choice.26-0940

[36] Erich Strohmaier Jack Dongarra Horst Simon Martin Meuer . 2019. 53rd Top 500 list. (2019). https://www.top500.org/lists/2019/06/

[37] Faucher-Giguère C.-A. Kaspi V. M. 2006. Birth and evolution of isolated radio pulsars. *Astrophys. J.* 643, 1 (2006), 332. DOI:http://dx.doi.org/10.1063/1.2900308

[38] Faulkner A. J. Stairs I. H. Kramer M. Lyne A. G. Hobbs G. B. et al. 2004. The Parkes Multibeam Pulsar Survey - V. Finding binary and millisecond pulsars. *Mon. Not. R. Astron. Soc.* 355, 1 (nov 2004), 147–158. DOI:http://dx.doi.org/10.1111/j.1365-2966.2004.08310.x

[39] Foster R. S. Backer D. C. 1990. Constructing a pulsar timing array. *Astrophys. J.* 361, 9 (sep 1990), 300. DOI:http://dx.doi.org/10.1086/169195

[40] Freire P. C. C. Wolszczan A. Berg M. V. D. Hessels J. W. T. 2007. A Massive Neutron Star in the Globular Cluster M5. *Astrophys. J.* 679 (2007), 10. DOI:http://dx.doi.org/10.1086/587832

[41] Frigo M. Johnson S. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (feb 2005), 216–231. DOI:http://dx.doi.org/10.1109/JPROC.2004.840301

[42] Gold T. 1968. Rotating Neutron Stars as the Origin of the Pulsating Radio Sources. *Nature* 218, 5143 (may 1968), 731–732. DOI:http://dx.doi.org/10.1038/218731a0

[43] Groth E. J. 1975. Probability distributions related to power spectra. *Astrophys. J. Suppl. Ser.* 29 (jun 1975), 285. DOI:http://dx.doi.org/10.1086/190343

[44] Gupta S. 2012. NEW TOP500 LIST: 4X MORE GPU SUPERCOMPUTERS. (2012). https://blogs.nvidia.com/blog/2012/07/02/new-top500-list-4x-more-gpu-supercomputers/

[45] Hankins T. H. 1971. Microsecond Intensity Variations in the Radio Emissions from CP 0950. *Astrophys. J.* 169 (1971), 487. DOI:http://dx.doi.org/10.1086/151164

[46] Helfand D. J. Manchester R. N. Taylor J. H. 1975. Observations of pulsar radio emission. III - Stability of integrated profiles. *Astrophys. J.* 198, June (jun 1975), 661. DOI:http://dx.doi.org/10.1086/153644

[47] Hewish A. Bell S. J. Pilkington J. D. H. Scott P. F. Collins R. A. 1968. Observation of a rapidly pulsating radio source. *Nature* 217, 5130 (nov 1968), 709–713. DOI:http://dx.doi.org/10.1038/224472b0

[48] Hobbs G. B. Bailes M. Bhat N. D. R. Burke-Spolaor S. Champion D. J. J. et al. 2009. Gravitational-wave detection using pulsars: status of the Parkes Pulsar Timing Array project. *Publ. Astron. Soc. Aust.* 26, 2 (mar 2009), 103–109. DOI:http://dx.doi.org/10.1071/AS08023

[49] Jankowski F. van Straten W. Keane E. F. Bailes M. Barr E. D. et al. 2018. Spectral properties of 441 radio pulsars. *Mon. Not. R. Astron. Soc.* 473, 4 (2018), 4436–4458. DOI:http://dx.doi.org/10.1093/mnras/stx2476

[50] Janssen G. H. Stappers B. W. Kramer M. Purver M. Jessner A. et al. 2008. European Pulsar Timing Array. In *AIP Conf. Proc.*, Vol. 983. AIP, 633–635. DOI:http://dx.doi.org/10.1063/1.2900317

[51] Johnston H. M. Kulkarni S. R. 1991. On the detectability of pulsars in close binary systems. *Astrophys. J.* 368 (feb 1991), 504–514. DOI:http://dx.doi.org/10.1086/169715

[52] Jouteux S. Ramachandran R. Stappers B. W. Jonker P. G. van der Klis M. et al. 2002. Searching for pulsars in close circular binary systems. *Astron. Astrophys.* 384, 2 (2002), 532–544. DOI:http://dx.doi.org/10.1051/0004-6361

[53] Kaspi V. M. Taylor J. H. Ryba M. F. 1994. High-precision timing of millisecond pulsars. 3: Long-term monitoring of PSRs B1855+09 and B1937+21. *Astrophys. J.* 428, 2 (jun 1994), 713. DOI:http://dx.doi.org/10.1086/174280

[54] Keane E. F. 2017. Pulsar Science with the SKA. *Proc. Int. Astron. Union* 13, S337 (sep 2017), 158–164. DOI:http://dx.doi.org/10.1017/S1743921317009188

[55] Keith M. J. Eatough R. P. Lyne A. G. Kramer M. Possenti A. et al. 2009. Discovery of 28 pulsars using new techniques for sorting pulsar candidates. *Mon. Not. R. Astron. Soc.* 395, 2 (may 2009), 837–846. DOI:http://dx.doi.org/10.1111/j.1365-2966.2009.14543.x

[56] Keith M. J. Jameson A. Van Straten W. Bailes M. Johnston S. et al. 2010. The High Time Resolution Universe Pulsar Survey - I. System configuration and initial discoveries. *Mon. Not. R. Astron. Soc.* 409, 2 (dec 2010), 619–627. DOI:http://dx.doi.org/10.1111/j.1365-2966.2010.17325.x

[57] Kramer M. Stairs I. H. Manchester R. N. McLaughlin M. A. Lyne A. G. et al. 2006. Tests of General Relativity from Timing the Double Pulsar. *Science (80-.).* 314, 5796 (2006), 97–102. DOI:http://dx.doi.org/10.1126/science.1132305

[58] Kramer M. Stappers B. W. 2015. Pulsar Science with the SKA. (jul 2015), 1–12. http://arxiv.org/abs/1711.01910http://arxiv.org/abs/1507.04423

[59] Kramer M. Wex N. 2009. The double pulsar system: A unique laboratory for gravity. *Class. Quantum Gravity* 26, 7 (2009). DOI:http://dx.doi.org/10.1088/0264-9381/26/7/073001

[60] Laidler C. B. Kuttel M. M. 2013. Detection of binary pulsars with GPU-accelerated sinusoidal Hough transformations. In *Astron. Soc. Pacific Conf. Ser.*, Vol. 475. 83. http://pubs.cs.uct.ac.za/archive/00000893/01/Laidler2012.pdf

[61] Lattimer J. M. Prakash M. 2004. The Physics of Neutron Stars. *Science (80-.).* 304, 5670 (2004), 536–542. DOI:http://dx.doi.org/10.1126/science.1090720

[62] Lazaridis K. Wex N. Jessner A. Kramer M. Stappers B. W. et al. 2009. Generic tests of the existence of the gravitational dipole radiation and the variation of the gravitational constant. *Mon. Not. R. Astron. Soc.* 400, 2 (2009), 805–814. DOI:http://dx.doi.org/10.1111/j.1365-2966.2009.15481.x

[63] Le Grand S. Götz A. W. Walker R. C. 2013. SPFP: Speed without compromise - A mixed precision model for GPU accelerated molecular dynamics simulations. *Comput. Phys. Commun.* 184, 2 (2013), 374–380. DOI:http://dx.doi.org/10.1016/j.cpc.2012.09.022

[64] Lee K. J. Guillemot L. Yue Y. L. Kramer M. Champion D. J. 2012. Application of the Gaussian mixture model in pulsar astronomy - pulsar classification and candidates ranking

for the Fermi 2FGL catalogue. *Mon. Not. R. Astron. Soc.* 424, 4 (2012), 2832–2840. DOI: http://dx.doi.org/10.1111/j.1365-2966.2012.21413.x

[65] Lee K. J. Stovall K. Jenet F. A. Martinez J. Dartez L. P. et al. 2013. PEACE: pulsar evaluation algorithm for candidate extraction – a software package for post-analysis processing of pulsar survey candidates. *Mon. Not. R. Astron. Soc.* 433, 1 (jul 2013), 688–694. DOI: http://dx.doi.org/10.1093/mnras/stt758

[66] Levin L. Armour W. Baffa C. Barr E. Cooper S. et al. 2017. Pulsar Searches with the SKA. *Proc. Int. Astron. Union* 13, S337 (2017), 171–174. DOI:http://dx.doi.org/10.1017/S1743921317009528

[67] Lorimer D. R. 2004. The Galactic Population and Birth Rate of Radio Pulsars. *Symp. - Int. Astron. Union* 218, circa 1997 (2004), 105–112. DOI:http://dx.doi.org/10.1017/s0074180900180726

[68] Lorimer D. R. 2008. Binary and Millisecond Pulsars. *Living Rev. Relativ.* 11, 1 (dec 2008), 8. DOI:http://dx.doi.org/10.12942/lrr-2008-8

[69] Lorimer D. R. Kramer M. 2004. *Handbook of pulsar astronomy*. Vol. 4. Cambridge University Press.

[70] Lyne A. G. Burgay M. Kramer M. Possenti A. Manchester R. N. et al. 2004. A Double-Pulsar System: A Rare Laboratory for Relativistic Gravity and Plasma Physics. *Science (80-.).* 303, 5661 (feb 2004), 1153–1157. DOI:http://dx.doi.org/10.1126/science.1094645

[71] Lyne A. G. Mankelow S. H. Bell J. F. Manchester R. N. 2000. Radio pulsars in Terzan 5. *Mon. Not. R. Astron. Soc.* 316, 3 (aug 2000), 491–493. DOI:http://dx.doi.org/10.1046/j.1365-8711.2000.03517.x

[72] Lyon R. J. Stappers B. W. Cooper S. Brooke J. M. Knowles J. D. 2016. Fifty years of pulsar candidate selection: From simple filters to a new principled real-time classification approach. *Mon. Not. R. Astron. Soc.* 459, 1 (2016), 1104–1123. DOI:http://dx.doi.org/10.1093/mnras/stw656

[73] Magro A. Karastergiou A. Salvini S. Mort B. Dulwich F. et al. 2011. Real-time, fast radio transient searches with GPU de-dispersion. *Mon. Not. R. Astron. Soc.* 417 (2011), 2642–2650. DOI:http://dx.doi.org/10.1111/j.1365-2966.2011.19426.x

[74] Manchester R. N. 2017. Millisecond Pulsars, their Evolution and Applications. *J. Astrophys. Astron.* 38, 3 (2017). DOI:http://dx.doi.org/10.1007/s12036-017-9469-2

[75] Manchester R. N. Hobbs G. Bailes M. Coles W. a. van Straten W. et al. 2013. The Parkes Pulsar Timing Array Project. *Publ. Astron. Soc. Aust.* 30 (2013), E017. DOI:http://dx.doi.org/10.1017/pasa.2012.017

[76] Manchester R. N. Hobbs G. B. Teoh A. Hobbs M. 2005. The Australia Telescope National Facility Pulsar Catalogue. *Astron. J.* 129, 4 (2005), 1993–2003. DOI:http://dx.doi.org/10.1109/URSIGASS.2014.6929987

[77] Manchester R. N. Lyne A. G. D'Amico N. Bailes M. Johnston S. et al. 1996. The parkes Southern pulsar Survey – I. Observing and data analysis systems and initial results. *Mon. Not. R. Astron. Soc.* 279, 4 (apr 1996), 1235–1250. DOI:http://dx.doi.org/10.1093/mnras/279.4.1235

[78] Manchester R. N. Taylor J. H. 1972. Parameters of 61 Pulsars. *Astrophys. Lett.* 10 (1972), 67–70.

[79] McLaughlin M. A. 2013. The North American Nanohertz Observatory for Gravitational Waves. *Class. Quantum Gravity* 30, 22 (nov 2013), 224008. DOI:http://dx.doi.org/10.1088/0264-9381/30/22/224008

[80] Michael Feldman . 2018. New GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500. (2018). https://www.top500.org/news/new-gpu-accelerated-supercomputers-change-the-balance-of-power-on-the-

[81] Mickaliger M. B. Lorimer D. R. Boyles J. McLaughlin M. A. Collins A. et al. 2012. Discovery of five new pulsars in archival data. *Astrophys. J.* 759, 2 (2012), 4–8. DOI:http://dx.doi.org/10.1088/0004-637X/759/2/127

[82] Middleditch J. Deich W. Kulkarni S. R. 1993. Searching for Millisecond Pulsars. In *Isol. pulsars Proc. Los Alamos Work. held Taos, New Mex. Febr. 23-28, 1992*. Cambridge Univ Pr, 372.

[83] Middleditch J. Priedhorsky W. C. 1986. Discovery of rapid quasi-periodic oscillations in Scorpius X-1. *Astrophys. J.* 306 (jul 1986), 230. DOI:http://dx.doi.org/10.1086/164335

[84] Morello V. Barr E. D. Cooper S. Bailes M. Bates S. et al. 2018. The High Time Resolution Universe survey XIV: Discovery of 23 pulsars through GPU-accelerated reprocessing. 14, November (2018), 1–14. DOI:http://dx.doi.org/10.1093/mnras/sty3328

[85] Nelder J. A. Mead R. 1965. A Simplex Method for Function Minimization. *Comput. J.* 7, 4 (jan 1965), 308–313. DOI:http://dx.doi.org/10.1093/comjnl/7.4.308

[86] Nethercote N. Seward J. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Not.* 42, 6 (2007), 89–100. DOI:http://dx.doi.org/10.1145/1250734.1250746

[87] Ng C. Champion D. J. Bailes M. Barr E. D. Bates S. D. et al. 2015. The High Time Resolution Universe Pulsar Survey - XII. Galactic plane acceleration search and the discovery of 60 pulsars. *Mon. Not. R. Astron. Soc.* 450, 3 (2015), 2922–2947. DOI:http://dx.doi.org/10.1093/mnras/stv753

[88] NVIDIA Corporation . 2010. *Nvidia GF100 Whitepaper*. Technical Report. 1–31 pages. http://www.nvidia.co.uk/object/IO

[89] NVIDIA Corporation . 2014. *NVIDIA NVLink High-Speed Interconnect: Application Performance (Whitepaper)*. Technical Report November. 19 pages.

[90] NVIDIA Corporation . 2015. Jen-Hsun On GeForce GTX 970. (2015). https://blogs.nvidia.com/blog/2015/02/24/gtx-970/

[91] NVIDIA Corporation . 2016a. *CUDA C Programming Guide 8.0*. Technical Report September. http://docs.nvidia.com/cuda/pdf/CUDA

[92] NVIDIA Corporation . 2016b. CUDA runtime API v8.0. (2016). https://docs.nvidia.com/cuda/cuda-runtime-api/

[93] NVIDIA Corporation . 2016c. *NVIDIA Tesla P100 Whitepaper*. Technical Report. 45 pages.

[94] NVIDIA Corporation . 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE: The World's Most Advanced Data Center GPU*. Technical Report August. 53 pages. http://www.nvidia.com/content/gated-pdfs/Volta-Architecture-Whitepaper-v1.1.pdf

[95] NVIDIA Corporation . 2019a. AmgX. (2019). https://developer.nvidia.com/amgx

[96] NVIDIA Corporation . 2019b. cuBLAS Library. (2019). https://docs.nvidia.com/cuda/cublas/

[97] NVIDIA Corporation . 2019c. CULA tools. (2019). http://www.culatools.com/

[98] NVIDIA Corporation . 2019d. cuRAND API. (2019). https://docs.nvidia.com/cuda/curand/

[99] NVIDIA Corporation . 2019e. cuSOLVER API. (2019). https://docs.nvidia.com/cuda/cusolver/

[100] NVIDIA Corporation . 2019f. The cuFFT API. (2019). https://docs.nvidia.com/cuda/cufft/

[101] NVIDIA Corporation . 2019g. The cuSPARSE API. (2019). https://docs.nvidia.com/cuda/cusparse/

[102] OpenACC_API . 2017. The OpenACC ™ Application Programming Interface. (2017). http://www.openacc.org/sites/default/files/OpenACC.2.0a

[103] Oppenheimer J. R. Volkoff G. M. 1939. On massive neutron cores. *Phys. Rev.* 55, 4 (1939), 374–381. DOI:http://dx.doi.org/10.1103/PhysRev.55.374

[104] Pacini F. 1968. Rotating Neutron Stars, Pulsars and Supernova Remnants. *Nature* 219, 5150 (jul 1968), 145–146. DOI:http://dx.doi.org/10.1038/219145a0

[105] Parent E. Kaspi V. M. Ransom S. M. Krasteva M. Patel C. et al. 2018. The Implementation of a Fast-folding Pipeline for Long-period Pulsar Searching in the PALFA Survey. *Astrophys. J.* 861, 1 (2018), 44. DOI:http://dx.doi.org/10.3847/1538-4357/aac5f0

[106] Pease M. C. 1968. An Adaptation of the Fast Fourier Transform for Parallel Processing. *J. ACM* 15, 2 (1968), 252–264. DOI:http://dx.doi.org/10.1145/321450.321457

[107] Pletsch H. J. Guillemot L. Allen B. Kramer M. Aulbert C. et al. 2012. Discovery of Nine Gamma-Ray Pulsars in Fermi Large Area Telescope Data Using a New Blind Search Method. *Astrophys. J.* 744, 2 (jan 2012), 105. DOI:http://dx.doi.org/10.1088/0004-637X/744/2/105

[108] Press W. H. Teukolsky S. Vetterling W. T. Flannery B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). Press Syndicate of the University of Cambridge. DOI:http://dx.doi.org/10.2307/1269484

[109] Press W. H. Teukolsky S. Vetterling W. T. Flannery B. P. 2007. *Numerical recipes The Art of Scientific Computing*. Cambridge University Press. DOI:http://dx.doi.org/10.2307/1269484

[110] Radhakrishnan V. Cooke D. J. 1969. Magnetic Poles and the Polarization Structure of Pulsar Radiation. *Astrophys. Lett.* 3 (1969), 225. http://adsabs.harvard.edu/cgi-bin/nph-bib

[111] Ransom S. M. 2012. Pulsars are cool. Seriously. *Proc. Int. Astron. Union* 8, S291 (aug 2012), 3–10. DOI:http://dx.doi.org/10.1017/S1743921312023046

[112] Ransom S. M. 2014. PRESTO. (2014). http://www.cv.nrao.edu/

[113] Ransom S. M. 2017. PRESTO Tutorial. (2017). http://www.cv.nrao.edu/

[114] Ransom S. M. Cordes J. M. M. Eikenberry S. S. S. S. 2003. A new search technique for short orbital period binary pulsars. *Astrophys. J.* 589, 2 (jun 2003), 911. DOI:http://dx.doi.org/10.1086/374806

[115] Ransom S. M. Eikenberry S. S. Middleditch J. 2002. Fourier Techniques for Very Long Astrophysical Time-Series Analysis. *Astron. J.* 124, 3 (sep 2002), 1788–1809. DOI: http://dx.doi.org/10.1086/342285

[116] Smith K. M. 2016. New algorithms for radio pulsar search. (2016), 1–16. http://arxiv.org/abs/1610.06831

[117] Smits R. Kramer M. Stappers B. W. Lorimer D. R. Cordes J. M. et al. 2008. Pulsar searches and timing with the SKA. *Astron. Astrophys.* 493, 3 (2008), 12. DOI:http://dx.doi.org/10.1051/0004-6361:200810383

[118] Smits R. Lorimer D. R. Kramer M. Manchester R. Stappers B. W. et al. 2009. Pulsar science with the Five hundred metre Aperture Spherical Telescope. *Astron. Astrophys.* 505, 2 (2009), 919–926. DOI:http://dx.doi.org/10.1051/0004-6361/200911939

[119] Staelin D. H. 1969. Fast folding algorithm for detection of periodic pulse trains. *Proc. IEEE* 57, 4 (1969), 724–725. DOI:http://dx.doi.org/10.1109/PROC.1969.7051

[120] Stairs I. H. 2004. Pulsars in Binary Systems: Probing Binary Stellar Evolution and General Relativity. *Science (80-.).* 304, 5670 (apr 2004), 547–552. DOI:http://dx.doi.org/10.1126/science.1096986

[121] Stappers B. W. 2013. The square kilometre array and the transient universe. *Philos. Trans. A. Math. Phys. Eng. Sci.* 371, April (2013), 20120284. DOI:http://dx.doi.org/10.1098/rsta.2012.0284

[122] Stappers B. W. Hessels J. W. Alexov A. Anderson K. Coenen T. et al. 2011. Observing pulsars and fast transients with LOFAR. *Astron. Astrophys.* 530 (2011). DOI:http://dx.doi.org/10.1051/0004-6361/201116681

[123] Stephen L. Moshier . 2016. Cephes Math Library. (2016). http://www.netlib.org/cephes/

[124] Stovall K. Lorimer D. R. Lynch R. S. 2013. Searching for Millisecond Pulsars: Surveys, Techniques and Prospects. *Class. Quantum Gravity* 30, 22 (2013), 16. DOI:http://dx.doi.org/10.1088/0264-9381/30/22/224003

[125] Taylor J. H. 1974. A sensitive method for detecting dispersed radio emission. *Astron. Astrophys. Suppl. Ser.* 15 (1974), 367. http://adsabs.harvard.edu/full/1974A

[126] Taylor J. H. Huguenin G. R. 1969. Two New Pulsating Radio Sources. *Nature* 221, 5183 (mar 1969), 816–817. DOI:http://dx.doi.org/10.1038/221816a0

[127] Taylor J. H. Weisberg J. M. 1982. A new test of general relativity - Gravitational radiation and the binary pulsar PSR 1913+16. *Astrophys. J.* 253 (1982), 908. DOI:http://dx.doi.org/10.1086/159690

[128] Taylor J. H. Weisberg J. M. 1989. Further experimental tests of relativistic gravity using the binary pulsar PSR 1913 + 16. *Astrophys. J.* 345 (1989), 434. DOI:http://dx.doi.org/10.1086/167917

[129] Torczon V. 1997. On the Convergence of Pattern Search Algorithms. *SIAM J. Optim.* 7, 1 (feb 1997), 1–25. DOI:http://dx.doi.org/10.1137/S1052623493250780

[130] Vaughan B. A. van der Klis M. Wood K. S. Norris J. P. Hertz P. et al. 1994. Searches for millisecond pulsations in low-mass X-ray binaries, 2. *Astrophys. J.* 435, 1 (nov 1994), 362. DOI:http://dx.doi.org/10.1086/174818

[131] Yusifov I. Küçük I. 2004. Revisiting the radial distribution of pulsars in the Galaxy. *Astron. Astrophys.* 422, 2 (2004), 545–553. DOI:http://dx.doi.org/10.1051/0004-6361:20040152

Appendix A

Software

The software developed in the course of this research is available in Github repository:

www.github.com/ChrisLaidler/presto

All our software is licensed under the GNU General Public License version 3.

Appendix B

Nvidia GPUs

Table B.1: Some details on Nvidia GPU hardware for a number of compute-specific GPUs over time. A number of GPUs have had a single configurable memory for both L1 cache and shared memory, this can be seen for CC 2.x, 3.x and 7.x

Year	Gen	CC	# SM	GPU	Chip	GFLOPS (FP32)	GFLOPS (FP64)	Bandwidth (GB/s)	Max Mem (GB)	L2 Cache	Cores FP32	Cores FP64	Cores FP16	LD/ST Units	# SFUs	Warp sched	L1 Cache (KB)	Shared Memory (KB)	Registers
2008	Tesla	1.3	16 SM	S870 GPU	G80	346	x	77	1.5	None	8	1	x	1	2	1	None	16	8192
2008	Tesla	1.3	30 SM	2200 S4	GT200	622	x	102	4	None	8	1	x	1	2	1	None	16	8192
2010	Fermi	2.0	16 SM	M2090 GPU	GF100	1331	665	148	6	768	32	16	x	16	4	2	64		32768
2012	Kepler	3.5	15 SMX	K20 GPU	GK110	3524	1175	208	12	1536	192	64	x	32	32	4	64		65536
2012	Kepler	3.7	16 SMX	K80	GK210	4368	1456	240	12	1536	192	64	x	32	32	4	128		131072
2015	Maxwell	5.2	5 SMM	M10	GM107	1322	165	332	12	2048	128	96	x	32	32	4	64	64	65536
2015	Maxwell	5.2	16 SMM	M60 GPU	GM204	4825	1508	320	24	2048	128	96	x	32	32	4	64	96	65536
2016	Pascal	6.0	56 SM	P100 GPU	GP100	10600	5300	549	16	4096	64	32	128	16	16	2	?	64	65536
2017	Volta	7.0	80 SM	V100 GPU	GV100	15700	7800	900	16	6144	64	32	128	32	16	4	128		65536

Appendix C

Timing of basic GPU functions

Table C.1: The number of GPU clock cycles required to complete a range of device functions

Function	GTX 770 CCpO	GTX 970 CCpO	GTX 1070 CCpO
mulf	1.2	1.0	1.0
muld	25.1	32.1	32.1
powf	113.6	115.6	93.0
__powf	13.1	10.2	9.2
sqrtf	54.9	28.3	20.2
__sqrtf	12.7	10.2	9.2
fabsf	2.3	2.0	2.1
fabsd	48.8	63.9	64.7
fmodf	13.0	19.1	13.8
fmod	72.7	95.8	97.0
modff	11.6	14.0	10.2
modf	47.5	64.9	64.7
sinf	31.9	29.0	25.2
__sinf	6.4	4.0	4.0
sind	425.8	471.9	477.7
sincosf	31.0	27.9	23.9
sincosd	549.7	678.1	687.8
__sincosf	6.1	4.0	4.1
sincospif	29.1	22.4	22.9
sqMod4f	18.6	9.9	9.0
sqMod4	86.2	96.3	97.0

Appendix D

The sqMod4 function

D.1 Single-precision `sqMod4` function

```
__device__ inline float sqMod4( float x )
{
  asm("{"
      " .reg .u32 r1;"                 // temp reg,
      " .reg .f32 f1, f2;"             // temp reg t1,
      " and.b32  r1, %1, 2139095040;" // Exponent bits
      " shr.b32  r1, r1, 23;"          // Shift to relevant spot
      " sub.s32  r1, 151, r1;"         // r1 = 24 - ( exp - 127 )
    -  Remove base - shift 24 bits for mantissa length
      " shr.b32  f2, %1, r1;"          // f2 = x  >> sft
      " shl.b32 f1, f2, r1;"           // f2 = f2 << sft
      " sub.f32 f2, %1, f1;"           // b = x - a
      " fma.rn.f32 f1, 2.0, f1, f2;"  // a = 2*a+b
      " mul.f32 %0, f1, f2;"           // a = a*b
      "}"
      : "=f"(x) : "f"(x));

  return x;
}
```

D.2 Double-precision `sqMod4` function

```
__device__ inline double sqMod4( double x )
{
  asm("{"
      " .reg .u32 r1;"                // temp reg,
      " .reg .f64 f1, f2;"            // temp reg t1,
      " .reg .u32 hi, lo;"            // temp reg,
      " mov.b64 {hi, lo}, %1;"
      " and.b32 r1, lo, 2146435072;"  // Exponent bits
      " shr.b32 r1, r1, 20;"          // Shift to relevant spot
      " sub.s32 r1, 1076, r1;"        // r1 = 53 - ( exp - 1023
   ) -  Remove base - shift 53 bits for mantissa length
      " shr.b64 f2, %1, r1;"          // f2 = x  >> sft
      " shl.b64 f1, f2, r1;"          // f2 = f2 << sft
      " sub.f64 f2, %1, f1;"          // b = x - a
      " fma.rn.f64 f1, 2.0, f1, f2;"  // a = 2*a+b
      " mul.f64 %0, f1, f2;"          // a = a*b
      "}"
      : "=d"(x) : "d" );

  return x;
}
```

Appendix E

Error and computation speed of GPU functions

Fresnel integrals

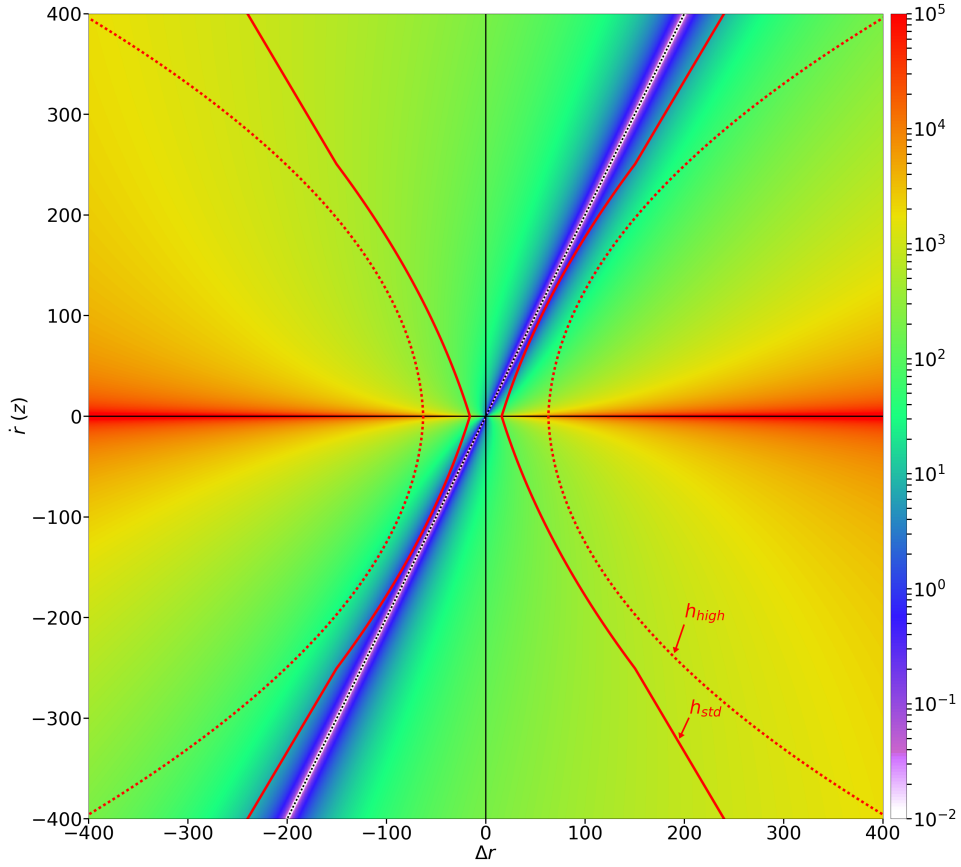


Figure E.1: Magnitude of the Fresnel factor in the Δr-$\dot{r}$ plane

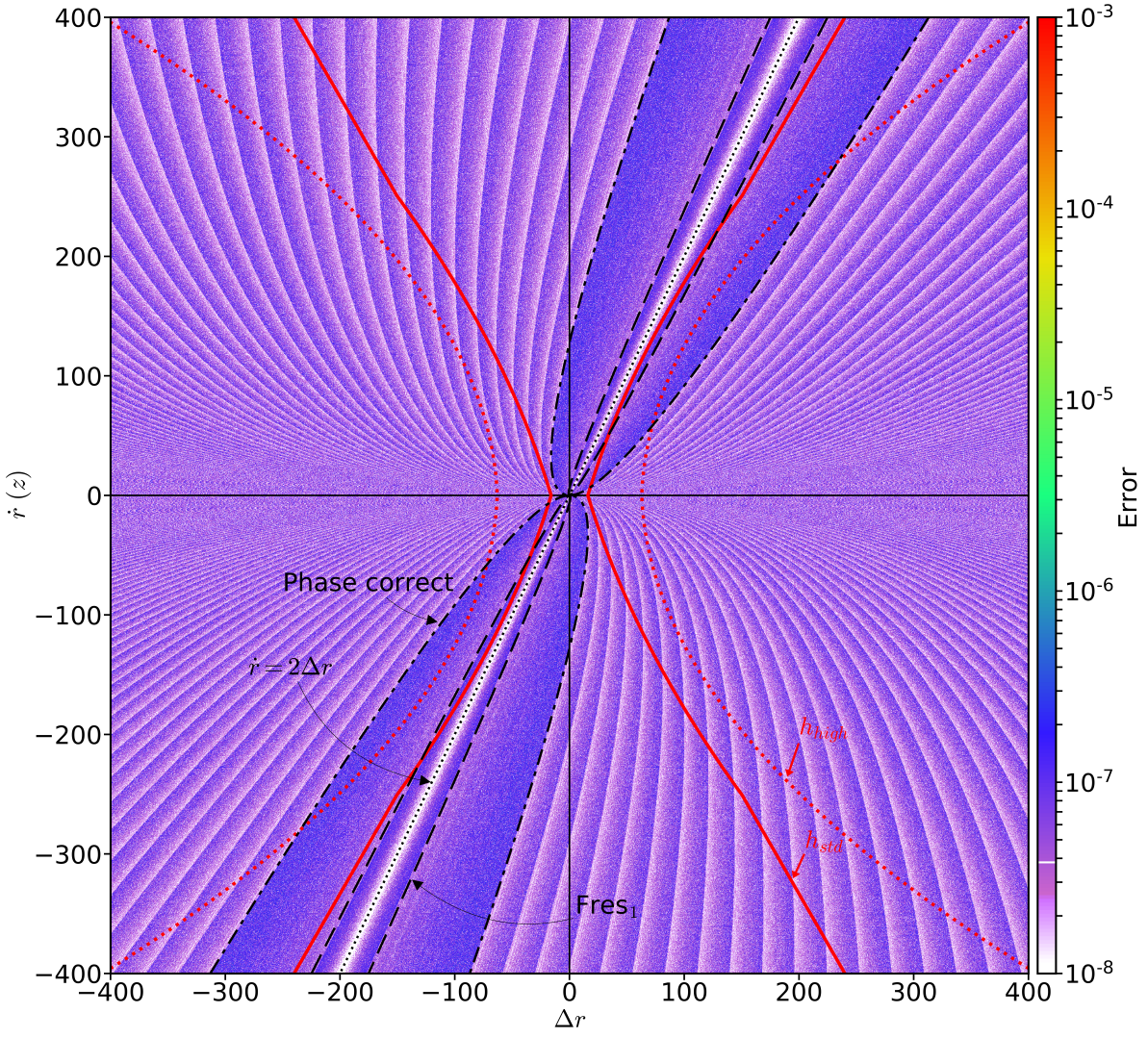


Figure E.2: Fresnel error - single-precision error of the Fresnel integrals of Y'. Showing bounds

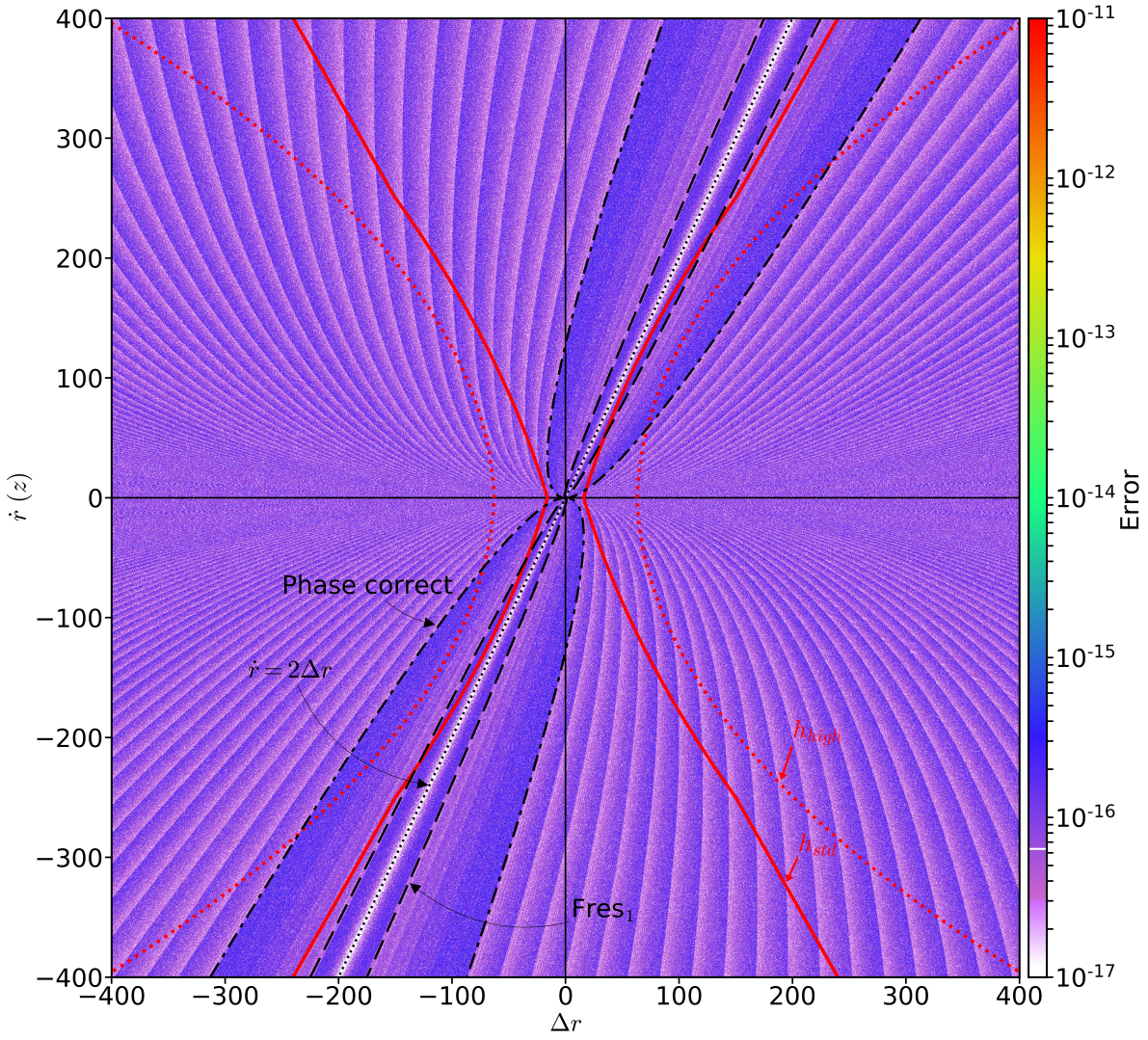


Figure E.3: Fresnel error - single-precision error of the Fresnel integrals of Y'

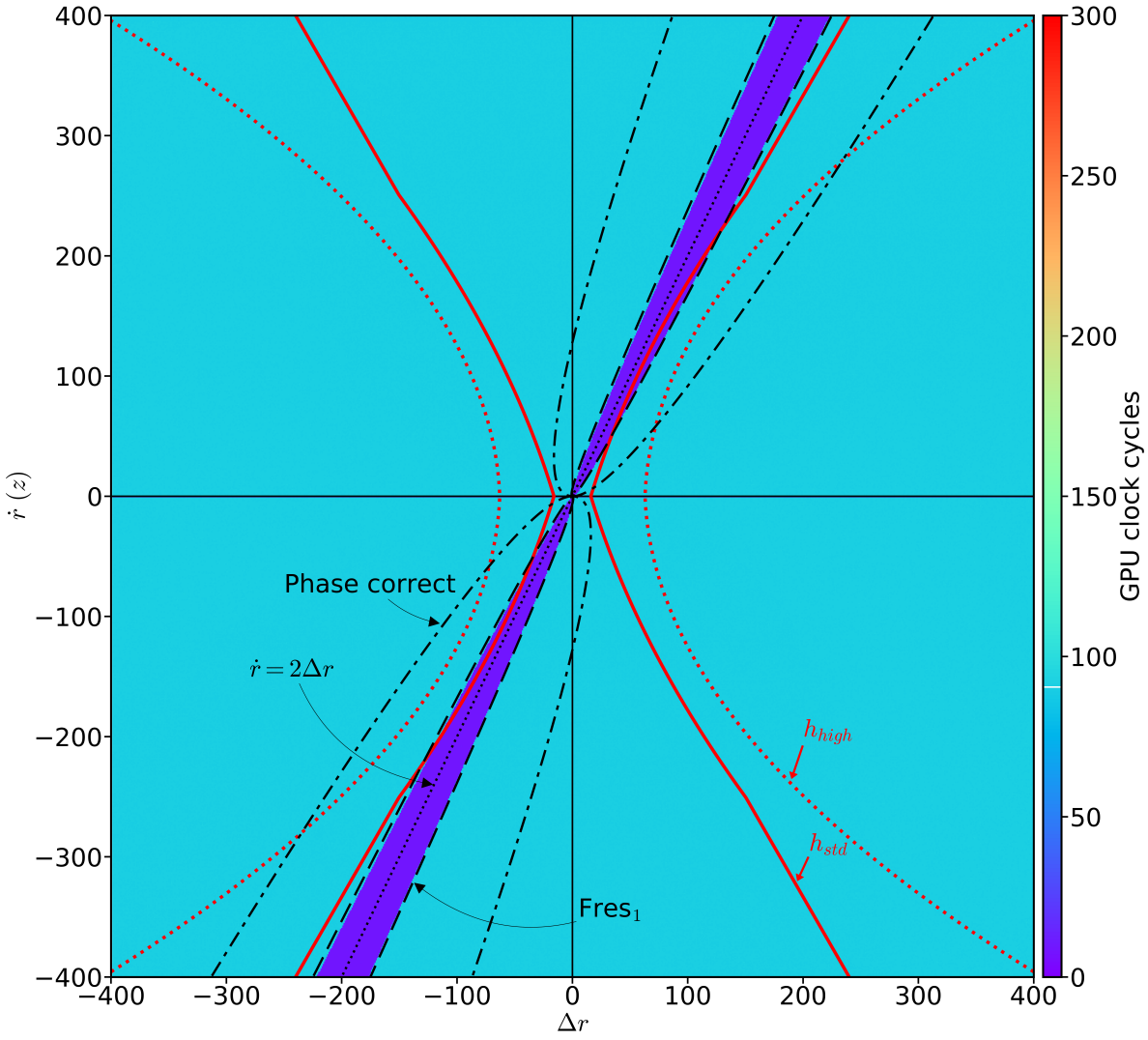


Figure E.4: Speed of the single-precision Fresnel function

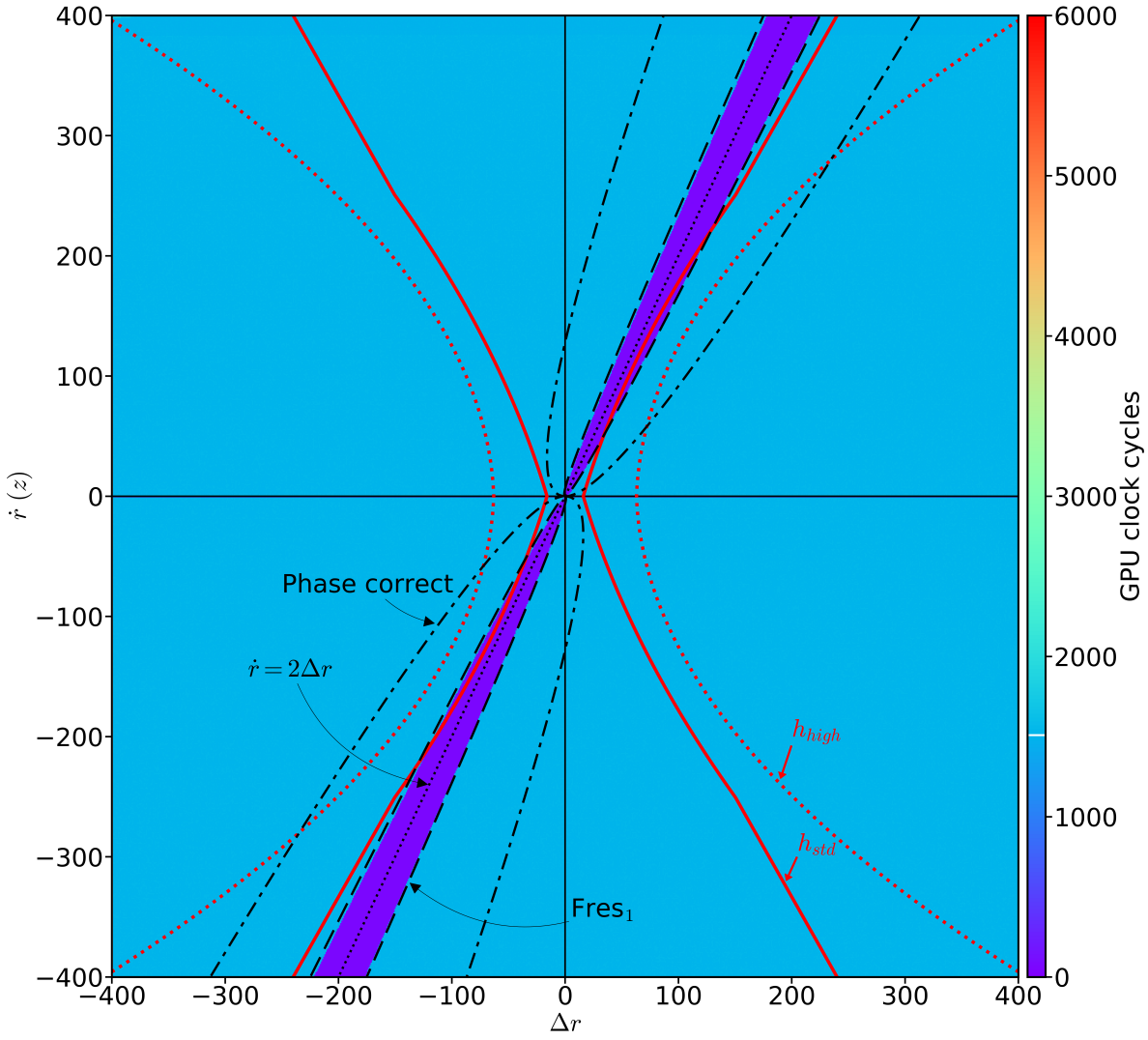


Figure E.5: Speed of the double-precision Fresnel function

Acceleration coefficients

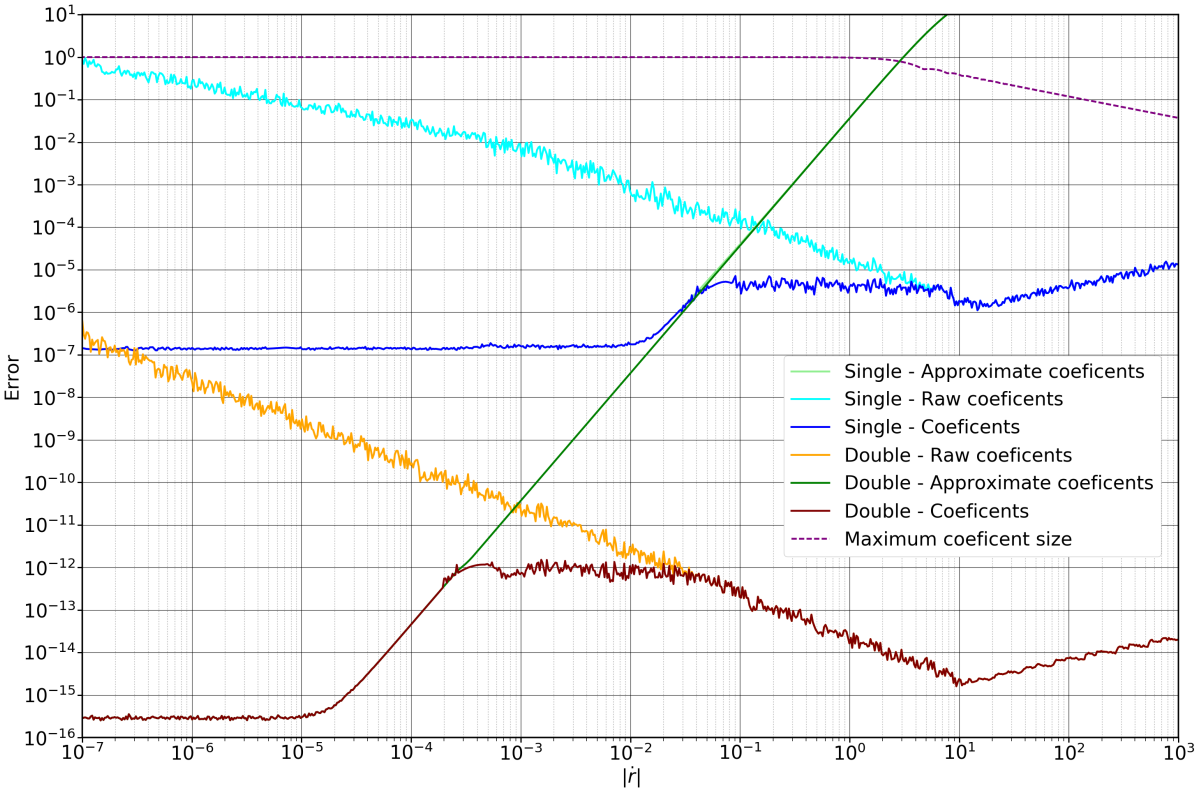


Figure E.6: Maximum error in acceleration coefficients.

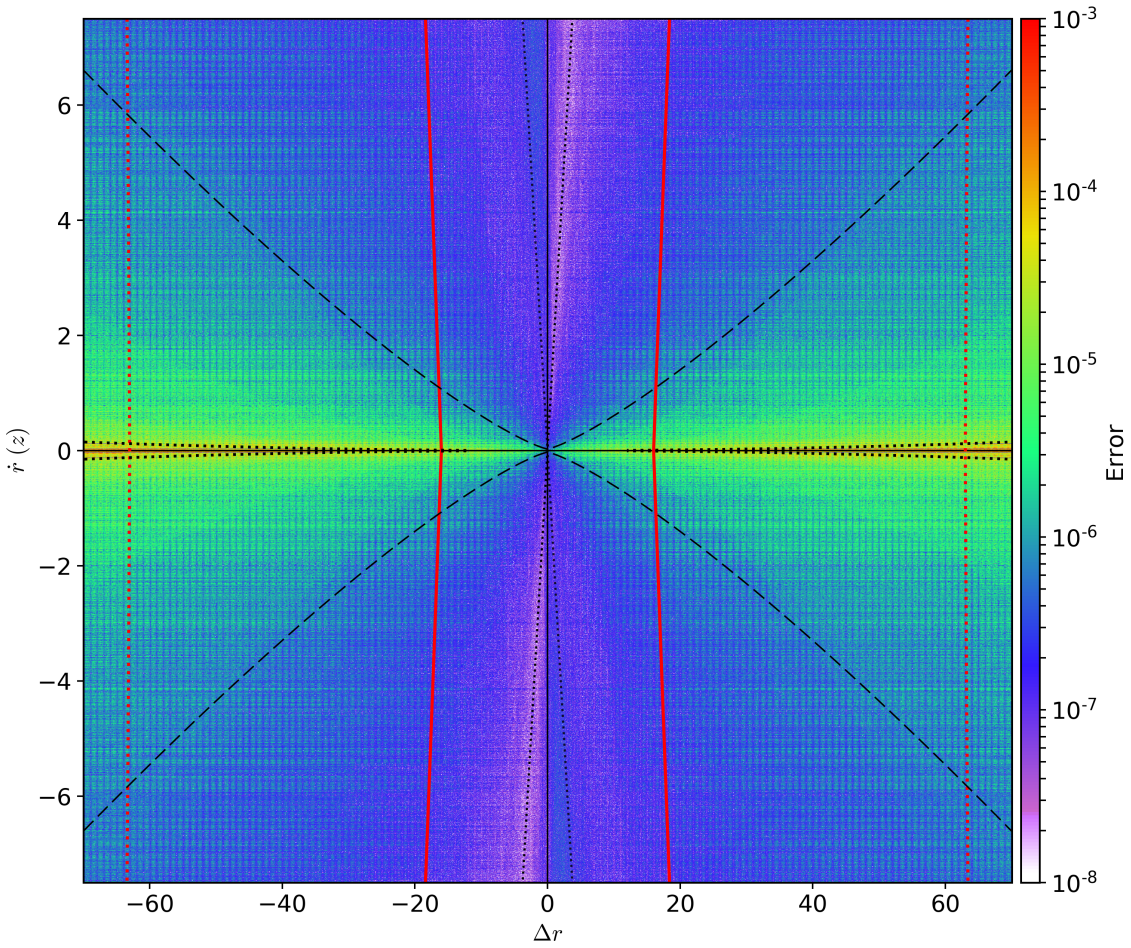


Figure E.7: Error of the raw acceleration coefficients calculated directly from Equation 6.4

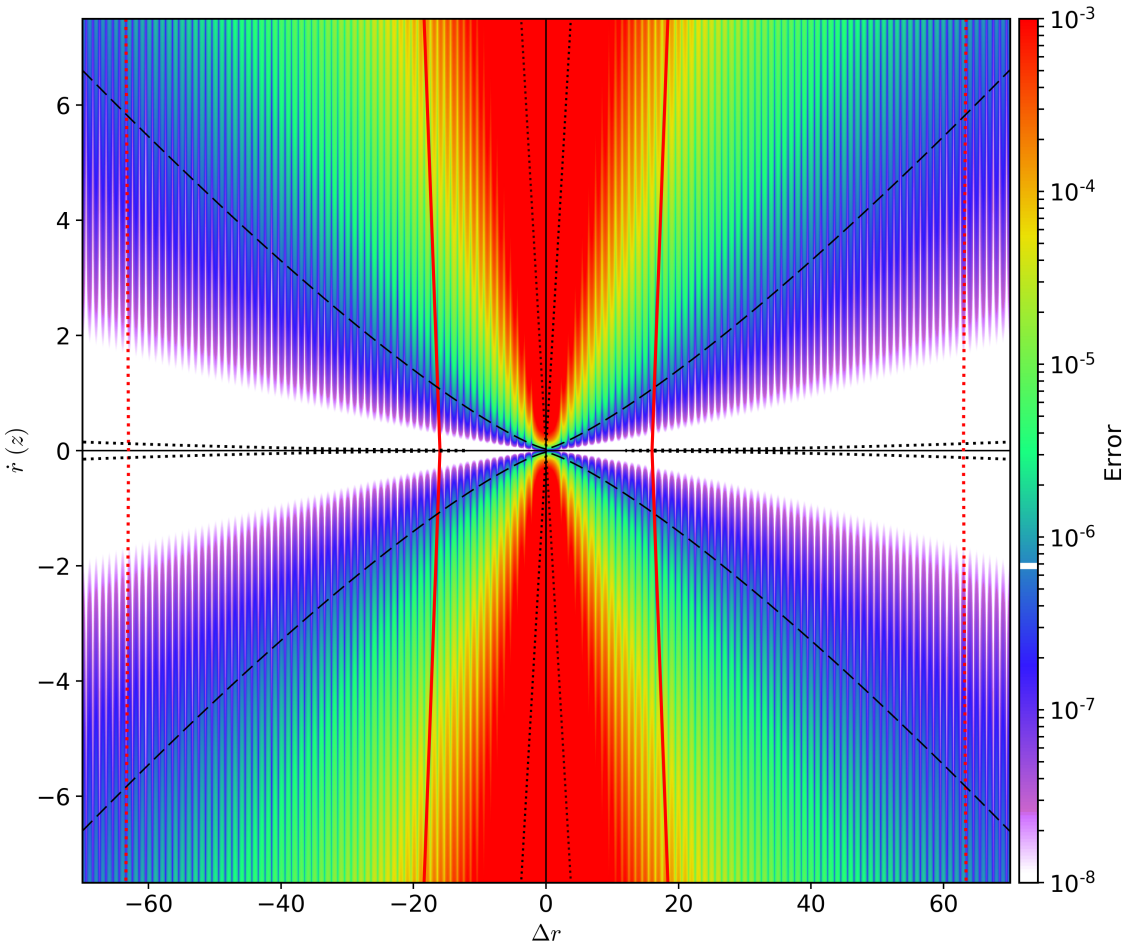


Figure E.8: Error of the approximated acceleration coefficients calculated from Equation 6.5

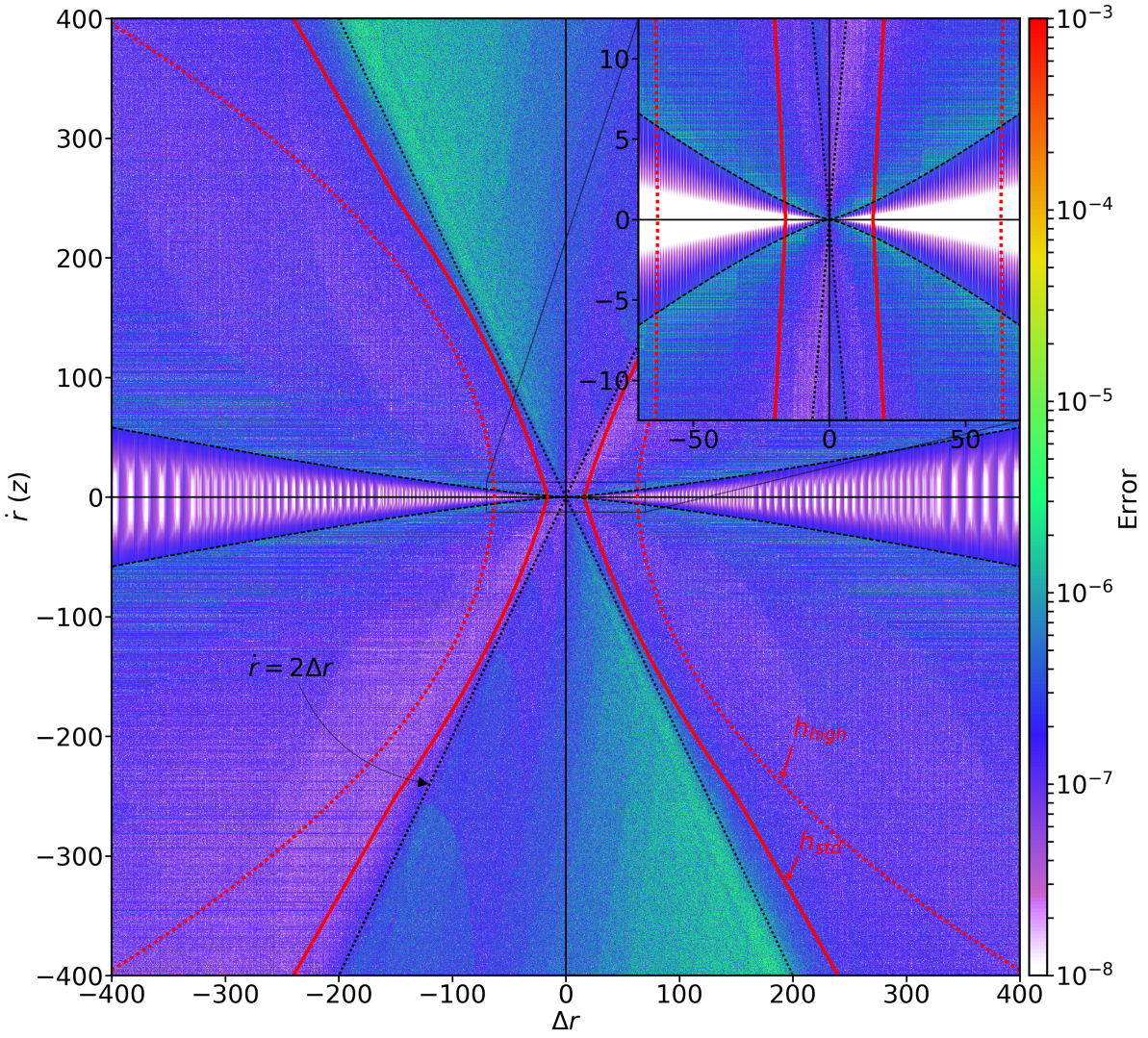


Figure E.9: Error in the single-precision generic acceleration coefficients

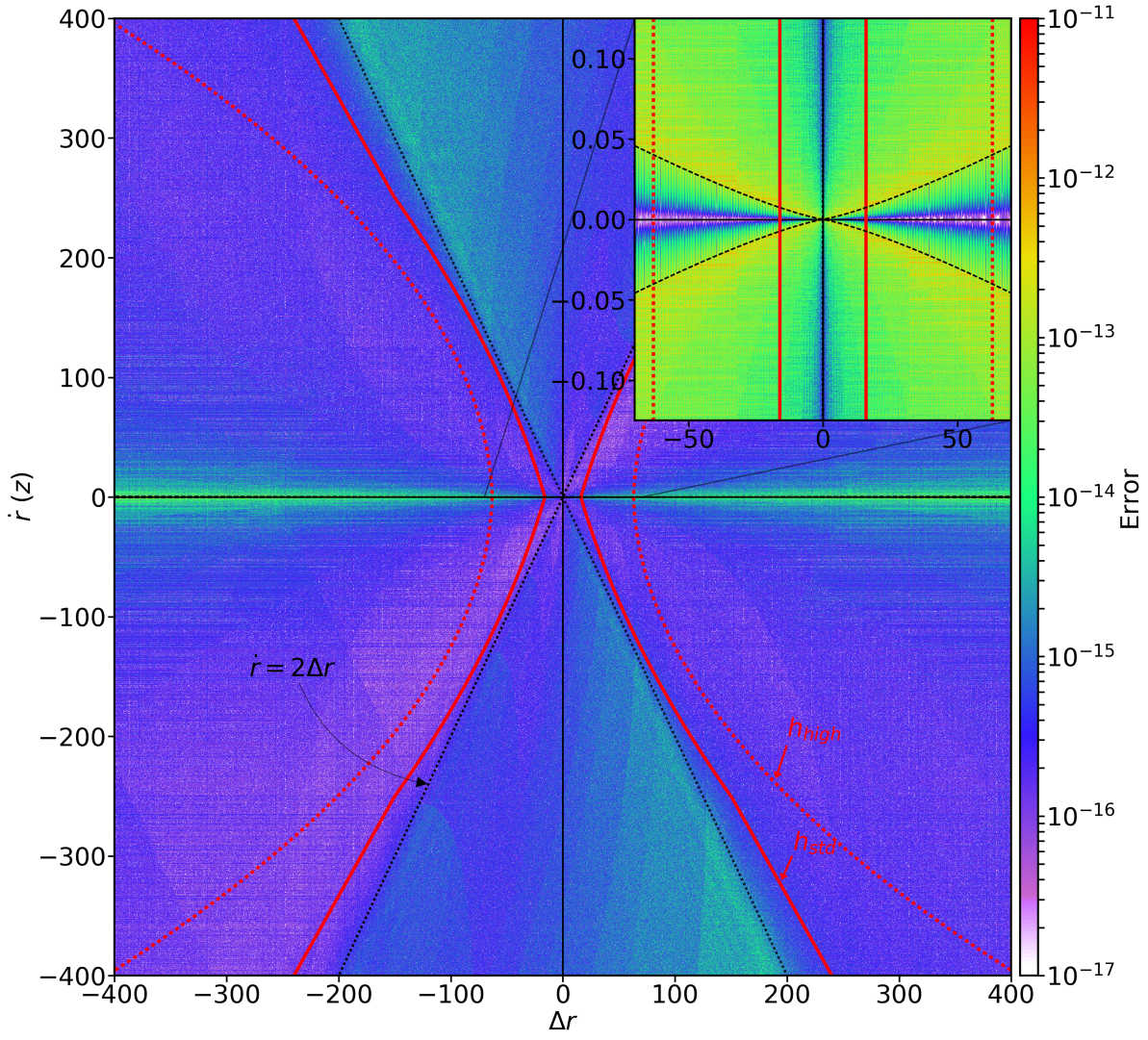


Figure E.10: Error in the double-precision generic acceleration coefficients

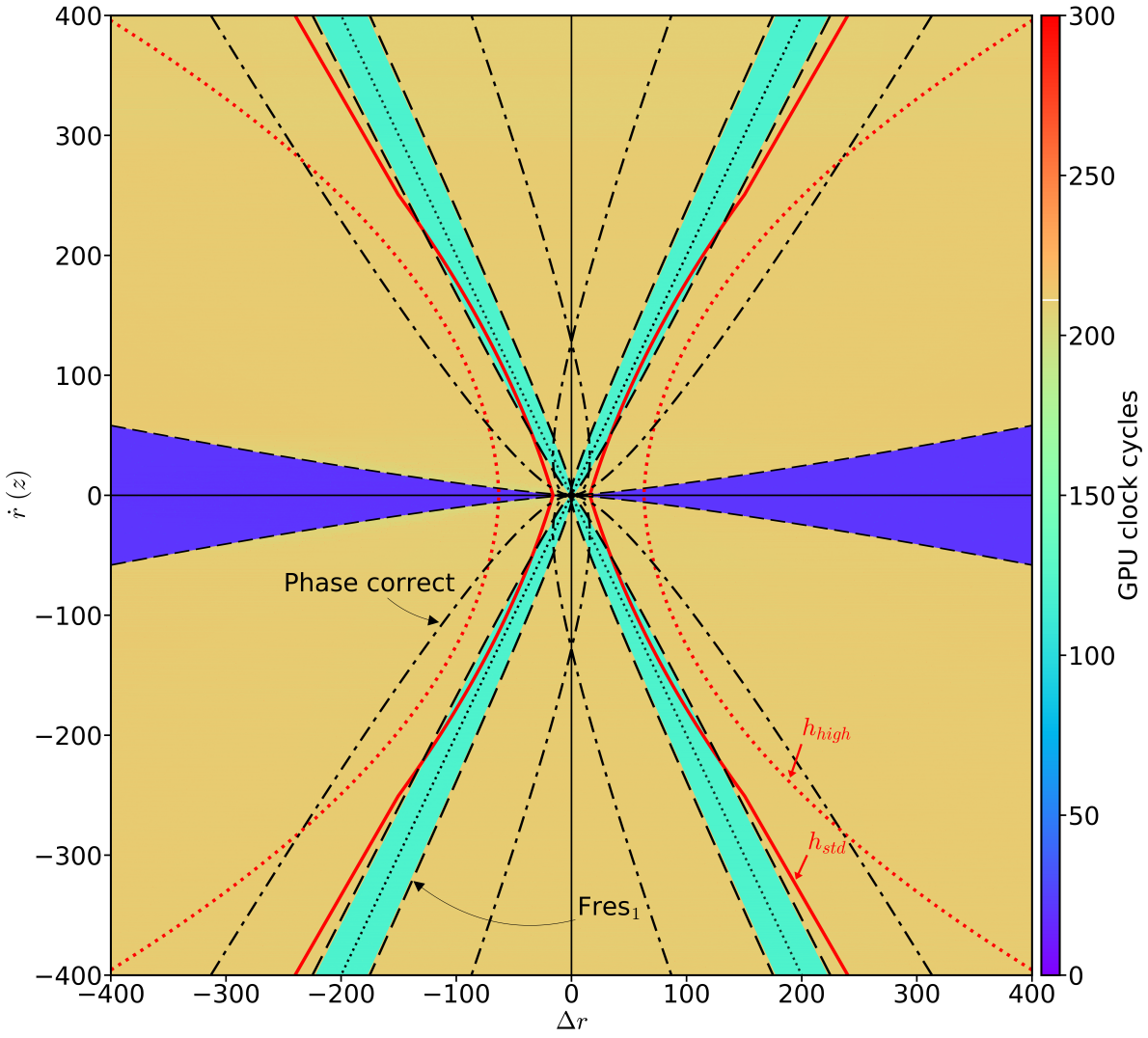


Figure E.11: Speed to calculate the single-precision generic acceleration coefficients

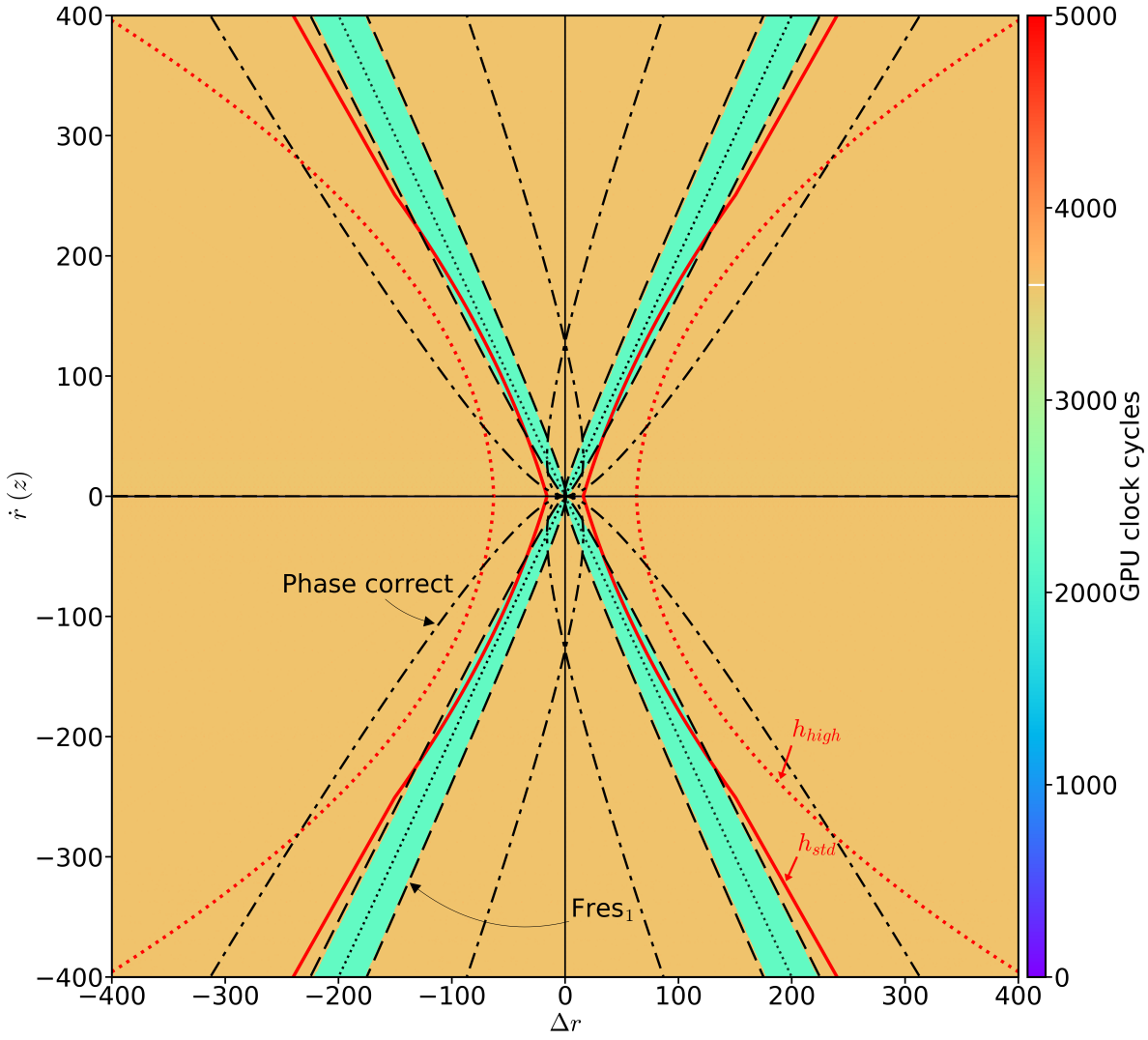


Figure E.12: Speed to calculate the double-precision generic acceleration coefficients